Ausgabe 06/2022

Java aktuell

# Java aktuell

iJUG Verbund
www.ijug.eu

**Java 19**
Das aktuelle Update im Detail

**Qualität**
Was macht Software hochwertig?

**Testing**
Wie gut sind meine Tests?

## QUALITÄT & TESTEN

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 06


#CloudLand2023

Eventpartner: Heise Medien

# Liebe Leserinnen und Leser,

wir starten wie gewohnt mit bunten Neuigkeiten rund um die Java-Community und deren Mitgliedern mit dem Java-Tagebuch und der Eclipse Corner. Auch die unbekannten Kostbarkeiten des SDK sind wieder mit von der Partie und widmen sich diesmal dem Revival des HTTP-Servers. Natürlich wollen und dürfen wir das aktuelle Java 19 Update nicht aus den Augen verlieren. Dieses nimmt Falk Sippach für uns wieder genauer unter die Lupe.

Den Auftakt in das Titelthema Qualität und Testing gibt Yves Schubert, der sich in seinem Artikel ab Seite 20 der Frage widmet, was genau Qualität und qualitativ hochwertige Software eigentlich sind und worauf es dabei ankommt.

Im Anschluss folgt eine wohl jedem Entwickelnden bekannte Methode zur Beurteilung der Qualität von Software: das Testen. Wiederum müssen auch diese Tests qualitativ hochwertig sein, um Fehler in der Anwendung aufdecken zu können. Nicolai Mainiero präsentiert uns als Lösung dafür das Mutation Testing, das Entwickelnden helfen soll, bessere Tests zu schreiben. Mit Thomas Ruhroth und Kai Schmidt bleiben wir beim Thema Testing – genauer: dem Sicherstellen von Architekturvorgaben mit ArchUnit. Die beiden Autoren nehmen uns mit auf eine Reise durch die verschiedenen Ebenen des ArchUnit-API und zeigen uns, welche Prüfungen auf welcher Ebene durchgeführt werden und wie diese untereinander kommunizieren.

Doch Qualität beinhaltet viel mehr als nur technische Aspekte wie Testing. Warum die Qualität einer Software maßgeblich auch durch das Team und dessen Kommunikation beeinflusst wird, beleuchten Janna Philipp und Anna Platschek in ihrem Artikel ab Seite 42.

Ihr habt auch ein spannendes Thema für die Java-Community auf Lager und habt Interesse, einen Artikel einzureichen? Traut euch! Informationen rund um die Zeitschrift und wie ihr euch als Autorin oder Autor einbringen könnt, findet ihr unter www.ijug.eu. Wir freuen uns, von euch zu hören!

Wir wünschen euch viel Spaß beim Lesen!
Eure

**Lisa Damerow**
Redaktionsleitung Java aktuell

# Inhalt


*Java 19: Was gibt's Neues?*


*Was ist eigentlich Softwarequalität?*


*Mutation Testing: Wie gut sind meine Tests?*


*Mehr als Testing: Welchen Einfluss hat das Team auf die Qualität von Software?*

# Das Java-Tagebuch

Andreas Badelt, stellv. Leiter der DOAG Java Community


## 18.07.2022

### VS Code mit vernünftiger Lombok-Unterstützung

Mit dem Juli-Release von Visual Studio Code hat Microsoft (nach eigenen Worten) die Probleme ihrer Extensions mit Lombok behoben, und gleichzeitig die Arbeit an der Community-initiierten Lombok Extension selbst übernommen, deren bisheriger Maintainer das Projekt wohl nicht weiter fortgeführt hat. Weitere Verbesserungen betreffen unter anderem Drag & Drop Support in der Explorer View und das Anlegen von „Function Breakpoints" (auch bei überladenen Methoden; bislang wurden nur Zeilen- und teilweise Exception-basierte Breakpoints unterstützt).

## 29.07.2022

### GraalVM 22.2

Version 22.2 der GraalVM ist da. Die größeren Veränderungen: Das Basis-JDK ist weiter modularisiert und dadurch verkleinert worden: Wer die JavaScript- oder LLVM-Runtimes benötigt (oder VisualVM), muss diese nachinstallieren. Der Native Image Builder ist in seinem Speicherbedarf reduziert worden – viele größere Native Images sollen sich jetzt mit 2 GB Heap Memory bauen lassen. Und es gibt nun zwei HeapDump-Optionen: „Dump and exit" – oder das Aktivieren der AllowVMInspection Option beim Build, sodass zur Laufzeit beispielsweise per SIGUSR1 HeapDumps angefordert werden können (es gibt noch weitere Möglichkeiten).

Extern zur GraalVM soll außerdem ein neues GitHub Repository für verschiedenste Komponenten „Reachability Metadata" zur Verfügung gestellt werden. Die Metadaten werden benötigt, wenn beim Bauen eines Native Images nicht alle benötigten Elemente per statischer Code-Analyse ermittelt werden können (etwa weil sie nur durch Reflection oder dynamische Proxies hinzugezogen werden). Das Fehlen dieser Elemente im Image kann dann zu Laufzeitfehlern führen. Genau das wird gelöst, indem dem Builder die entsprechenden Reachability-Metadaten mitgegeben werden. Der Mechanismus ist nicht neu, aber bislang war die mühsame Erstellung der Daten ein individueller Zeitvertreib, und erforderte auch tiefere Kenntnis der Komponenten. Genau hier soll das neue Repository Abhilfe schaffen. Ziel ist es, möglichst die Projekte, die die Komponenten selbst erstellen, auch zur Veröffentlichung der Reachability-Daten zu motivieren. Ein gutes Dutzend Projekte, zum Beispiel Tomcat Embedded, Netty oder h2, sind bereits dabei, aber es ist noch ein weiter Weg *[1]*.

## 05.08.2022

### Adoptium News

Das Adoptium-Projekt will in seinem Markplatz künftig Warnungen für Binaries anzeigen, die älter als 180 Tage alt und damit aus Security-Sicht anfällig sind, da sie seit zwei Quartalen keine aktuellen Patches haben. Bislang gibt es keine Bedenken, also wird es wohl demnächst umgesetzt.

Ein weiteres Thema ist aktuell, dass RedHat die OpenJDK „Upstream-Builds" einstellen möchte, die noch unter AdoptOpenJDK veröffentlicht werden. Alle Nutzer sollen auf Temurin umstellen. Das klitzekleine Problem: Die Upstream Builds sind die Basis für die offiziellen OpenJDK Docker Images. Also wird wohl erst ein Migrationsplan mit den Docker-Hub-Leuten besprochen werden.

## 08.08.2022

### Java und datenorientierte Programmierung

Schon vor einigen Wochen hat der der Chef-Architekt der Sprache Java (im Gegensatz zur Plattform), Brian Goetz, einen sehr interessanten Artikel über daten-orientierte Programmierung auf infoQ geschrieben, ich bin jetzt darüber gestolpert. Darin beschreibt er, wie sich objekt- und datenorientierte Programmierung nicht ausschließen, sondern situationsbezogen auch in Java ergänzen. Wie die Beiträge von Project Amber zum JDK, insbesondere Sealed Classes, Records und Pattern Matching, datenorientierte Programmierung in Java dramatisch vereinfachen *[2]*.

## 09.08.2022

### Java Records und Jakarta EE – Fortsetzung der Diskussion

Parallel zum Artikel von Brian Goetz – oder scheinbar auch zum Teil davon motiviert – ist die Diskussion um Records und (Persistenz-) Frameworks wieder aufgeflammt, die Anfang des Jahres schon mal im Tagebuch vorkam. Es geht um Anforderungen an NoArg-Konstruktoren, die Records ja per Definition nicht bieten (mit Ausnahme des eher sinnlosen leeren Records). Kurz gesagt, es gibt keine konkreten Pläne, aber ein paar Ideen, und wer sich für das Thema näher interessiert, dem sei dieser Thread ans Herz gelegt *[3]*.

## 10.08.2022

### MicroProfile Spezifikationen angenommen

In Vorbereitung auf das nächste MicroProfile-Release sind in den letzten Tagen eine Reihe von Spezifikationen vom „Steering Committee" angenommen worden, die dann in einigen Wochen offiziell freigegeben werden: Reactive Streams Operators 3.0, JWT 2.1 und OpenAPI 3.1. Ersteres ist lediglich eine technische Anpassung an Jakarta EE 9.1 und die neuen Package-Namen, was aber einen Breaking Change bedeutet, daher 3.0. Für JWT sind diese Anpassungen bereits mit 2.0 erfolgt, bei 2.1 gibt es eine Reihe kleinerer Neuerungen, beispielsweise die konfigurative Definition eines maximalen Token-Alters; oder eine heftig debattierte Konfigurationsoption, um den Algorithmus zur Entschlüsselung des symmetrischen „Content Encryption Keys" serverseitig festzulegen *[4]*. Bei OpenAPI 3.1 sieht es ähnlich aus, neue Features sind beispielsweise „extensions"-Attribute für die meisten Annotationen, „additionalProperties" für @Schema, und eine Reihe von Ergänzungen und Klarstellungen bei den Security-Definitionen.

20.08.2022

## Log4Shell – fix es schnell (und andere Träumereien)

15 Minuten nach Bekanntwerden einer kritischen Schwachstelle starten Hacker typischerweise die ersten Scans, um verwundbare Systeme zu finden, so steht es in einem Bericht von Palo Alto Networks, basierend auf eigenen Untersuchungen zu mehr als 600 „Incidents" ihrer Kunden in den letzten 12 Monaten. Das Ausnutzen bekannter Schwachstellen liegt dabei mit 31 % der erfolgreichen Attacken weit vor anderen Methoden wie „brute force" oder anderweitig erbeuteten Passwörtern, nur noch übertroffen von Phishing (der Mensch ist halt doch immer noch die größere Schwachstelle). Die immer kürzer werdende Reaktionszeit macht Sorgen – noch viel mehr aber ist es die komplett ausbleibende Reaktion bei vielen Systemverantwortlichen. Der Lerneffekt, unter anderem aus Log4Shell, ist nicht groß genug, viele nutzen nicht nur Software ohne aktuelle Patches, sondern Software, die längst „end of life" ist und gar keine Patches mehr erhält (und wissen es häufig vermutlich nicht mal). Das im Februar gegründete US Cyber Safety Review Board (CSRB) hat Log4Shell übrigens als „endemisch" erklärt. Es sei davon auszugehen, dass die (prinzipiell ja geschlossene) Lücke uns noch weitere zehn Jahre beschäftigen wird.

Immerhin scheint der Aspekt Firmware (und Lieferketten) mehr in den Fokus der Security-Verantwortlichen zu rücken, wie ein aktueller heise-Artikel berichtet. Hier herrschte in der Vergangenheit wohl auch zu viel „Urvertrauen in Hardware".

24.08.2022

## Jakarta EE WG – NEC wird Mitglied

Ein weiteres Mitglied in der Jakarta EE Working Group: Die in IT-Maßstäben altehrwürdige japanische NEC Corporation (gegründet 1899) ist hinzugekommen. Mit WebOTX bringt der Tech-Mischkonzern einen ebenso altehrwürdigen Application Server mit, der zumindest schon für Java EE 7 zertifiziert ist, und in Zukunft auch „Jakarta EE compliant" werden soll.

24.08.2022

## MicroProfile 6.0 mit Telemetry 1.0 und Metrics 5.0

Die MicroProfile (MP) Telemetry 1.0 und Metrics 5.0 Spezifikationen sind angenommen worden, neben der Angleichung an Jakarta EE 10 die Hauptbestandteile des für Oktober oder November angekündigten MicroProfile 6.0 Releases. Gleichzeitig gab es eine Abstimmung darüber, wie mit der OpenTelemetry-Implementierung zu verfahren ist, die standardmäßig in Telemetry 1.0 deaktiviert sein soll. Es wurde entschieden, den „Original-Switch" aus dem OpenTelemetry-Projekt zu verwenden (otel.experimental.sdk.enabled, das „experimental" soll noch verschwinden), um dessen Tracing, Propagation etc. explizit zu *aktivieren*.

Wäre es nicht standardmäßig deaktiviert, würde es ohne weitere Aktion parallel zum MicroProfile-eigenen OpenTracing laufen, so dieses genutzt wird. OpenTracing wird nämlich mit MicroProfile 6.0 aus dem „Umbrella" entfernt, aber erst mal noch als optionale Einzelspezifikation weitergeführt; in Zukunft löst sich dieser Konflikt dann ohnehin auf, wenn die Implementierungen sich auf OpenTelemetry fokussieren.

07.09.2022

## Jakarta EE 10 Release steht an

Nun ist es so weit. Mit ein paar Monaten Verspätung wird Java EE 10 nächste Woche freigegeben werden. Nach einer weiteren Last-Minute-Verzögerung sieht es nicht mehr aus. Was bringt Release 10 in den Standard? Auf jeden Fall endlich OpenId-Connect-Unterstützung (mit Annotations) in Jakarta Security 3.0. Und in Jakarta Concurrency 3.0 kommt ein modernes asynchrones API dazu, das die existierenden asynchronen Ansätze zusammenführt und erweitert. Die Nutzung von CDI über die Einzelspezifikationen hinweg ist deutlich konsistenter und umfassender geworden. Außerdem muss jede Einzelspezifikation jetzt ihr eigenes JPMS-Modul definieren (oder mehrere). Neben Web und Full Profile gibt es in Release 10 ein Core Profile, das auf Microservices, „Serverless" und Ähnliche ausgerichtet ist. Java 11 muss jetzt von allen Implementierungen unterstützt werden.

Das ist noch nicht alles. Aber Markus Karg wird in seiner „Eclipse Corner" sicher einiges zum Release und seiner Geschichte schreiben.

09.09.2022

## Zum Schluss: Java 19

Java 19 kommt kurz nach Redaktionsschluss der Java aktuell raus, aber ich gehe kein hohes Risiko ein, wenn ich sage, dass es beim 20. September bleiben wird. Ein Feature – die Record Patterns, die das vereinfachte Arbeiten mit instanceof auch auf Records erweitern – hatte ich bislang nicht erwähnt. Es ist aber auch erst mal eine Preview.

Interessanter wäre der Blick auf Java 20, aber da tut sich momentan noch nicht viel. Ein heißer Kandidat ist der aktuell im Inkubator-Status befindliche JEP 429: „Extent-Local Variables". Hier geht es um das Teilen von unveränderlichen („immutable") Daten in und zwischen Threads, die speziell für große Anzahlen (virtueller) Threads die bevorzugte Variante gegenüber Thread Local Variablen darstellen. Der JEP kommt logischerweise aus dem Projekt Loom mit seinen „Fibers". Ansonsten gibt es ja in Release 19 eine ganze Reihe Previews/Inkubator-Features, von denen hoffentlich das eine oder andere in 20 schon als stabil deklariert wird. Das nächste Longterm Support Release ist ja 21 im Herbst nächsten Jahres.

Ach ja, für die „Late Adopters" (der Engländer sagt auch „laggards") sei noch erwähnt, dass Java 7 seit Ende Juli nun aber wirklich komplett aus dem Support raus ist, auch aus dem zahlungspflichtigen erweiterten Support. Ok, Oracle nennt die Phase jetzt ganz amerika-


nisch „sustaining support", macht aber in der Definition selbst ziemlich klar, dass das nichts mit „sustainable" und wenig mit „Support" zu tun hat, da es noch nicht mal mehr Sicherheits-Fixes gibt.

## Referenzen

[1] *https://github.com/oracle/graalvm-reachability-metadata*
[2] *https://www.infoq.com/articles/data-oriented-programming-java/*
[3] *https://www.eclipse.org/lists/jakarta.ee-community/msg02976.html*
[4] *https://github.com/eclipse/microprofile-jwt-auth/issues/289*

**Andreas Badelt**
stellv. Leiter der DOAG Java Community
*andreas.badelt@doag.org*

Andreas Badelt ist stellvertretender Leiter der DOAG Java Community. Er ist seit dem Jahr 2001 ehrenamtlich im DOAG e.V. aktiv, zunächst als Co-Leiter der SIG Development und später der SIG Java. Seit 2015 ist er stellvertretender Leiter der neugegründeten Java Community innerhalb der DOAG. Beruflich hat er seit dem Jahr 1999 als Entwickler und Architekt für deutsche und globale IT-Beratungsunternehmen gearbeitet und ist seit dem Jahr 2016 als freiberuflicher Software-Architekt unterwegs („www.badelt.it").

# Markus' eclipse-Corner

Mit dem Schreiben dieser Ausgabe der Eclipse Corner habe ich mal wieder bis auf den letzten Drücker gewartet. Diesmal absichtlich, und wie ihr euch sicher denken könnt, heißt der Grund, wie so oft, auch dieses Mal: Jakarta EE. Es mag sein, dass Jakarta EE 10 veröffentlicht ist, wenn ihr diese Zeilen lest. Möglicherweise aber auch nicht. Denn noch vorgestern, im regelmäßigen Platform Call der Jakarta EE Working Group (eine offene Telefonkonferenz, in der jeder Fragen an die Gruppe stellen kann), konnte oder wollte sich kein Beteiligter dazu verpflichten, ein definitives Release-Datum zu nennen. So langsam entwickelt sich das mit großem Tam-Tam angekündigte, erste echte Feature-Release des Java-EE-Nachfolgers zu „Vaporware". In besagtem Meeting wurde mitgeteilt, dass es keinerlei offene Punkte mehr gibt (worauf warten wir dann?), dass man Großartiges geleistet hätte (was genau, blieb eher nebulös) und dass alle Beteiligten ganz, ganz toll sind (lassen wir es mal dahingestellt; vermutlich gibt es das Sprichwort mit dem stinkenden Eigenlob in den USA nicht). Nur selbst applaudiert haben sie sich nicht, das hat mich etwas überrascht (Riten dieser Art sind in den USA teilweise üblich). Super, dann ist ja alles gut! Es wurde eine lange Liste mit Dingen geschrieben, die super geklappt hätten, mit „wir haben re-

gelmäßig miteinander telefoniert" als einer der Top-Leistungen. Es wurde eine kurze Liste mit Dingen geschrieben, die man noch besser machen könnte. Gut. Echte Features zu liefern, in nennenswerter Anzahl, mit sinnvoller Nutzung, oder gar zeitgerecht, war leider nicht dabei. Schade.

Von einer Gruppe, die sich selbst angesichts dieser Glanzleistung immer noch ganz toll findet und zu Selbstkritik offenbar nicht ernsthaft fähig ist, brauchen wir auch in Zukunft sicherlich nichts anderes zu erwarten. Angeblich war der Grund für die neuerlichen Verzögerungen, dass man wohl „vergessen" hätte, GlassFish auf mehr als nur einem Java-Release zu testen, obwohl ja sowohl Java SE 11 als auch 17 als Basis von Anfang an versprochen waren (und sonst ja eigentlich nicht sonderlich viel). Fehler passieren, und Vergesslichkeit ist sicherlich kein Verbrechen. Aber wieso dies gleich mehrere Monate an Verzögerung verursachen soll bei einem Projekt, an dem so viele Personen in Vollzeit beteiligt waren – das klingt doch eher fadenscheinig. Die Wahrheit ist: So viele sind es halt nicht, und schon gar nicht in Vollzeit. Die GitHub-Statistik verrät ja jedem, wie es wirklich aussieht.

Nach dem, was wir mit Jakarta EE 8, 9 und 9.1 erlebt hatten, nach dem, was ich euch in den letzten Monaten und Jahren an dieser Stelle berichtet hatte, war denn allen Ernstes etwas anderes zu erwarten? Vermutlich nicht. Mir wird gerne vorgeworfen, ich würde alles schlechtreden. Mit Sicherheit ist da etwas dran – Provokation und Polarisation sind ja durchaus gezielte journalistische Stilmittel. Trotzdem ist, zumindest Stand heute, weder Jakarta EE 10 freigegeben noch ein zertifiziertes, kommerziell supportetes Produkt auf dem Markt (GlassFish setzt ja sicherlich niemand ernsthaft produktiv ein). Selbst wenn ich an dieser Stelle also in Lobhudelei ausbrechen würde, würde das nichts an dieser Tatsache ändern. Und meine persönliche Meinung ist: Auch Jakarta EE 11 käme dann sicher nicht schneller, zeitgerechter oder mit mehr Features. Ohne Kritik gibt es keinen Grund, sich zu verbessern. Kritik ist etwas Positives, das zu höherer Leistung anspornt. Wir können ja gerne mal den umgekehrten Weg gehen und allen dazu gratulieren, dass Jakarta EE 10 noch nicht da ist, grundlos noch weitere Wochen nicht da sein wird, und der Jakarta EE Working Group auch noch Beifall dafür klatschen. Vielleicht kommt Jakarta EE 11 ja dann sogar ausnahmsweise früher raus als geplant. Dummerweise glaube ich das nur leider nicht. Und, liebe Leser, ganz ehrlich: ihr doch auch nicht, oder?

Nun könnte ich an dieser Stelle einen guten Freund zitieren: „Wir machen es wie immer, nur noch doller." Gut, dann warten wir halt weiter auf Jakarta EE. Mache ich aber nicht. Stattdessen schaue ich selbst in den Spiegel und sage: „Hey, was haben WIR eigentlich für Jakarta EE 10 getan?" Ja, einige haben tatsächlich etwas geleistet, und denen gebührt unser aller Dank. Ganz im Ernst und ohne Schalk im Nacken. Die meisten von uns aber, und mit „uns" meine ich jeden deutschsprachigen Nutzer von Jakarta EE, glänzten doch leider eher mit Abwesenheit. Insofern brauchen „wir" uns ja auch nicht zu beklagen. Von nichts kommt bekanntlich nichts. So ist das nun mal in jedem Verein.

Mit Freuden werde ich also (hoffentlich) in der nächsten Ausgabe über die Neuerungen von Jakarta EE 10 berichten, denn tatsächlich sind ja durchaus einige tolle Features enthalten – so sie denn endlich veröffentlicht sind. Bis dahin jedoch bleibt mit nur, wie in jeder Ausgabe, auch an dieser Stelle zu mahnen, zu bitten und zu betteln: Leute, wenn ihr wollt, dass Jakarta EE eine Zukunft hat, dann engagiert euch endlich. Denn einige der Großen der Branche fahren ihr Engagement bereits deutlich herunter, zugunsten von nativen (sprich: nicht portablen) Lösungen à la Quarkus, Helidon & Co. Wer nicht wieder in die Vendor-Lock-in-Falle geraten will, tut also gut daran, endlich in die Puschen zu kommen. Der iJUG, und auch ich selbst, helfen euch gerne mit euren ersten Contributions. Open Source lebt von Engagement – unserem!

**Markus Karg**
*markus@headcrashing.eu*

Markus Karg ist Entwicklungsleiter eines mittelständischen Softwarehauses sowie Autor, Konferenzsprecher und Consultant. JAX-RS hat der Sprecher der Java User Group Goldstadt von Anfang an mitgestaltet, zunächst als freier Contributor, seit JAX-RS 2.0 als Mitglied der Expert Groups JSR 339 und JSR 370.

# Unbekannte Kostbarkeiten des SDK Heute: Revival des HTTP-Servers

**Bernd Müller, Ostfalia**

*Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.*

Wir wollen in unserer heutigen Kolumne der unbekannten Kostbarkeiten kein unbekanntes API vorstellen, sondern freuen uns, dass Oracle ein solches von uns bereits 2014 beschriebenes API mit entsprechendem Marketing 2022 weitaus mehr publik gemacht hat, als uns das 2014 wahrscheinlich gelungen ist: den HTTP-Server.

## Der HTTP-Server

Bereits 2014 haben wir in der Java aktuell 03-2014 den im JDK enthaltenen HTTP-Server vorgestellt *[1]*. Wer diese Ausgabe nicht mehr im Regal stehen hat, aber einen Blick in den Artikel werfen will, kann dies mit der PDF-Version *[2]* tun. Aufgrund der Online-Verfügbarkeit verzichten wir auch auf das Darstellen der damals geschriebenen Beispielanwendung. Das Package der wenigen Implementierungsklassen des HTTP-Servers ist `com.sun.net.httpserver`. Wir haben die Beispielanwendung dieses für IT-Begriffe doch recht alten Artikels mit Java 18 übersetzt und ausgeführt. Alles funktionierte problemlos wie damals. Unser Dank geht an die sehr hoch gehaltenen Kompatibilitätsansprüche des JDK.

## Der neue HTTP-Server

Der JEP 408 *[3]* ist mit „Simple Web Server" überschrieben und hat das Ziel, einen einfachen Web-Server als Kommandozeilenwerk-

zeug bereitzustellen. Als Einsatzbereiche werden Prototyping, Ad-hoc-Programmierung und Testzwecke, vor allem im Ausbildungsumfeld, genannt. Das genannte Ziel wurde erstmals mit Java 18 durch das Werkzeug jwebserver umgesetzt. Die mediale Aufmerksamkeit durch das Oracle-Marketing und die Java-Community war beachtlich.

Die Implementierung von `jwebserver` basiert letztendlich auf denselben Klassen, die wir bereits 2014 verwendet haben, und wurde durch wenig zusätzlich Neues realisiert. Die Anzahl der Klassen im Package wuchs lediglich von 13 auf 16. Eine der neuen Klassen, die Klasse `SimpleFileServer` ist die Implementierung hinter `jwebserver`. Die beiden anderen neuen Klassen sind `HttpHandlers`, eine Fabrik für Standardimplementierungen des Interface `HttpHandler`, sowie Request, ein Interface zur Modellierung eines Request. Die bereits bestehenden 13 Klassen wurden lediglich auf der Ebene des JavaDoc überarbeitet, die jeweilige APIs blieben bis auf ganz wenige Ausnahmen bestehen.

## Geschichtsforschung

Nachdem der HTTP-Server wiederbelebt (obwohl er nie tot war) und mit einem Kommandozeilenwerkzeug einer breiten Öffentlichkeit präsentiert wurde, interessiert uns seine Geschichte. Bis zur Java-Version 5 war der HTTP-Server nicht im JDK enthalten. In den Versionen 6, 7 und 8 ist er als Byte-Code, seit Version 9 auch als Quellcode im JDK enthalten. Dies liegt wahrscheinlich an der zu dieser Zeit realisierten Umstellung von Closed Source zu Open Source und damit einhergehenden Lizenzklärungen begründet. Die im vorherigen Abschnitt erwähnten Änderungen am Code des Packages wurden im Rahmen des JEP 408 mit Java 18 realisiert.

## Zusammenfassung

Das mit Java 18 veröffentlichte und im JEP 408 spezifizierte Kommandozeilenwerkzeug `jwebserver` basiert auf einer Implementierung, die bereits seit Java 6 im JDK enthalten ist und von uns bereits im Jahr 2014 als unbekannte Kostbarkeit veröffentlicht wurde. Durch Oracles Marketing sowie die mittlerweile in unzähligen Blogs, Zeitschriftenartikeln und Konferenzvorträgen übliche Vorstellung neuer JDK-Features dürfte nun allseits bekannt sein, dass das JDK diesen primitiven Web-Server enthält, der jedoch auch ohne Kommandozeilenwerkzeug, wie in unserem ursprünglichen Artikel beschrieben, verwendet werden kann.

Für die jüngeren Entwickler unter den Lesern möchte ich anmerken, dass nicht alles, was als neu angepriesen wird, auch wirklich neu ist. Mein mittlerweile erreichtes Alter möge mir diese oberlehrerhafte Anmerkung erlauben ;-)

## Referenzen


[1] Bernd Müller: Unbekannte Kostbarkeiten des SDK – Heute: HTTP-Server. In Java aktuell 03-2014.

[2] *https://www.pdbm.de/files/java-aktuell-2014-3-http-server.pdf*

[3] JEP 408: Simple Web Server: *https://openjdk.org/jeps/408*



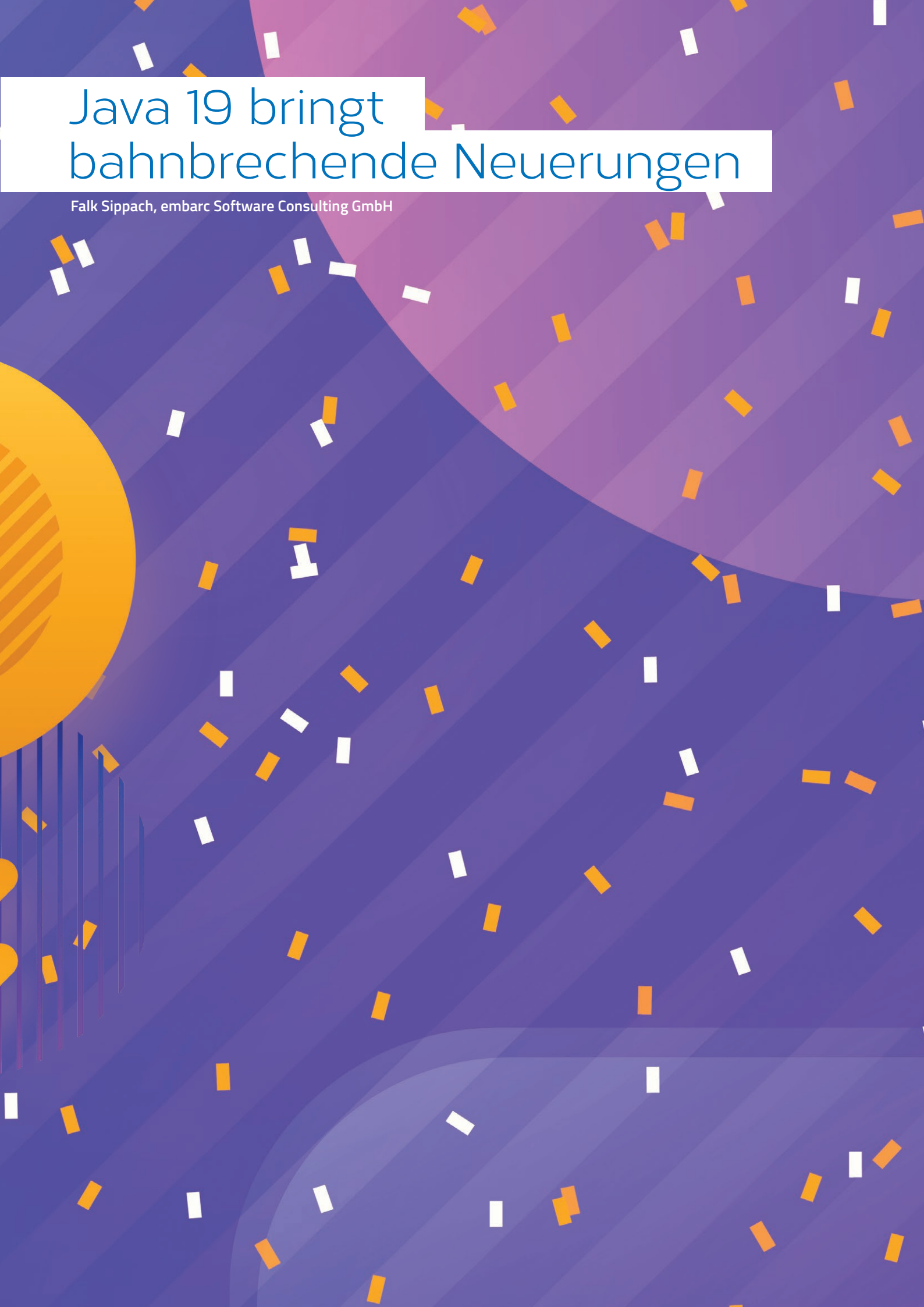
**Bernd Müller**
Ostfalia
*bernd.mueller@ostfalia.de*


Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.


JAVA
19

# Java 19 bringt bahnbrechende Neuerungen

Falk Sippach, embarc Software Consulting GmbH



*Auf den ersten Blick scheint in Java 19 nicht so viel passiert zu sein. Natürlich geht es beim Pattern Matching for Switch weiter und neu hinzugekommen sind die Record Patterns. Das Vector API und das Foreign Function & Memory API sind ebenfalls mit von der Partie. Wenn man aber noch etwas genauer hinschaut, bahnt sich eine Revolution an. Die Virtual Threads und die darauf aufbauende Structured Concurrency sind zwei wegweisende Features. Sie werden in den nächsten Jahren die Art, wie wir in Java mit Nebenläufigkeit umgehen, gehörig umkrempeln. Doch alles schön der Reihe nach, wir werfen in diesem Artikel einen Blick auf alle Updates des JDK 19.*

Für einen ersten Überblick ist wie immer die Projektseite des OpenJDK *[1]* ein guter Startpunkt. Dort sind diesmal zwar nur 7 JEPs (JDK Enhancement Proposals) gelistet, die haben es allerdings teilweise in sich und werden uns auch in den nächsten Releases noch weiter beschäftigen:

- JEP 405: Record Patterns (Preview)
- JEP 422: Linux/RISC-V Port
- JEP 424: Foreign Function & Memory API (Preview)
- JEP 425: Virtual Threads (Preview)
- JEP 426: Vector API (Fourth Incubator)
- JEP 427: Pattern Matching for switch (Third Preview)
- JEP 428: Structured Concurrency (Incubator)

Wie so oft, sind nicht alle Themen für uns Java-Entwickler direkt relevant. Beispielsweise wurde durch die zunehmende Verbreitung von RISC-V-Hardware mit dem JEP 422 ein Port für diese Rechner-Architektur zur Verfügung gestellt. Auch das Vector API (JEP 426), bereits seit Java 16 regelmäßig enthalten, ist bereits zum vierten Mal im Inkubator-Status mit von der Partie. Dabei geht es im Übrigen nicht um die bereits seit Java 1.0 existierende Klasse `java.util.Vector`. Vielmehr sollen die Mittel ins JDK, die die modernen SIMD-Rechnerarchitekturen mit Vektorprozessoren in Zukunft bieten. Single Instruction Multiple Data (SIMD) lässt mehrere Prozessoren gleichzeitig unterschiedliche Daten verarbeiten. Durch die Parallelisierung auf Hardware-Ebene verringert sich beim SIMD-Prinzip der Aufwand bei rechenintensiven Schleifen.

Ebenfalls wieder im „Recall" ist das Foreign Function & Memory API (JEP 424). Es bietet die Möglichkeit, aus Java nativen C-Code aufzurufen und auf Speicher außerhalb der JVM zuzugreifen. Die Älteren unter uns erinnern sich noch an das Java Native Interface (JNI). Doch das war sehr aufwendig zu verwenden, ziemlich fragil und auch nicht typsicher. Ziel des neuen API ist es, den Implementierungsaufwand um 90 % zu reduzieren und die Leistung um mindestens Faktor 4 zu beschleunigen. Im Gegensatz zum Vector API hat das JEP 424 diesmal den Inkubator-Status verlassen und ist jetzt als erste Preview verfügbar. Da ist also in einem der nächsten Java-Releases eine finale Version in Sicht. Doch natürlich kann es zwischen den Preview-Phasen auch weiterhin Änderungen geben.

Bevor wir gleich zu den aus Entwicklersicht spannenderen Themen kommen, sei für die vielen weiteren kleineren Aktualisierungen auf die Release Notes *[2]* verwiesen. Diese zeigen eine vollumfängliche Liste aller Änderungen in Java 19. So gibt es auch in der Klassenbibliothek des JDK wieder diverse Updates. Zum Beispiel wurden die öffentlichen Konstruktoren der Klasse Locale als „deprecated" markiert. Stattdessen sollen nun die neuen statischen Factory-Methoden `Locale.of()` verwendet werden. Dadurch wird sichergestellt, dass es pro `Locale`-Konfiguration nur eine Instanz gibt. Solche Informationen und überhaupt Änderungen an der Klassenbibliothek kann man im Übrigen recht gut über den Java-Almanac nachverfolgen *[3]*.

## Pattern Matching

Bereits seit einigen Jahren wird im Rahmen von Project Amber *[4]* Stück für Stück an der Einführung von Pattern Matching gearbeitet. Dazu waren diverse Änderungen in der Sprache selbst notwendig. Los ging es bereits im JDK 12 mit den Switch Expressions. Ab Version 14 folgten die Type Patterns (Pattern Matching for instanceof) und Records. Sealed Classes wurden im JDK 15 eingeführt und mit 17 kam „Pattern Matching for switch" als erste Preview hinzu. Letzteres war überraschend. In einem LTS-Release hatte man bisher keine Preview-Stände erwartet. Vielmehr sollten im OpenJDK 17 die Arbeiten der letzten fünf Major-Releases konsolidiert und finalisiert werden. Doch die aktuelle Update-Politik von Java scheut sich nicht davor, jederzeit auch noch nicht finale Zwischenstände neuer Features einfließen zu lassen. Ziel solcher Preview-Funktionen ist das kontinuierliche Ausliefern nützlicher Funktionen und vor allem das frühzeitige Feedback zu diesen Neuerungen aus der Community.

Beim Pattern Matching geht es darum, bestehende Strukturen gegen Muster abzugleichen. Ein Pattern ist dabei eine Kombination aus einem Prädikat (das auf die Zielstruktur passt) und einer Menge von Variablen innerhalb dieses Musters. Diesen Variablen werden bei passenden Treffern die entsprechenden Inhalte zugewiesen und damit extrahiert. Die Intention des Pattern Matching ist die Destrukturierung von Objekten, also das Aufspalten in die Bestandteile und das Zuweisen in einzelne Variablen zur weiteren Bearbeitung. Diese Destrukturierung schauen wir uns gleich bei den Record Patterns noch genauer an.

Pattern Matching for Switch ist als JEP 427 bereits zum dritten Mal als Preview dabei. Und diesmal gibt es tatsächlich nochmal eine größere Änderung an der Syntax. Die zuvor eingeführten Guarded Patterns, die Type Patterns konkretisieren können, werden durch sogenannte `when`-Clauses ersetzt. Das soll die Lesbarkeit erhöhen und Mehrdeutigkeiten bei der bisherigen Syntax mit dem booleschen And („&&") vermeiden. Das Schlüsselwort „when" ist in dem Fall kontextbezogen und kann außerhalb von `case`-Zweigen weiterhin als Name für Variablen oder Methoden verwendet werden. Ein schönes Beispiel dafür, wie man in Java bei Änderungen auf Abwärtskompatibilität achtet. *Listing 1* zeigt ein Beispiel, in dem nach großen Dreiecken mit einer Fläche von über 100 Einheiten gefiltert wird.

```
class Shape {
}
class Rectangle extends Shape {
}
class Triangle extends Shape {
int calculateArea() {
return 1000;
     }
}

Shape s = new Triangle();
switch (s) {
  case Triangle t when t.calculateArea() > 100 -> System.out.println("Large triangle");
  case Triangle t -> System.out.println("Small triangle");
  default -> System.out.println("Not a triangle");
}
```

*Listing 1*

```
record Point(int i, int j) {}
enum Color { RED, GREEN, BLUE; }

static void typeTester(Object o) {
switch (o) {
case null -> System.out.println("null");
case String s -> System.out.println("String");
case Color c -> System.out.println("Color with " + c.values().length + " values");
case Point p -> System.out.println("Record class: " + p.toString());
case int[] ia -> System.out.println("Array of ints of length" + ia.length);
default -> System.out.println("Something else");
}
}
```

*Listing 2*

Wichtig ist hier natürlich die Reihenfolge der `case`-Zweige. Die speziellen Kriterien (mit `when-Clause`) müssen vor den allgemeinen (nur das Type Pattern) stehen. Andersherum würde es der Compiler aber auch mit einem Fehler quittieren, weil der Spezialfall sonst nie aufgerufen werden würde.

Ansonsten hat sich zur letzten Preview nicht viel verändert und vermutlich biegt dieses Feature im JDK 20 oder spätestens 21 auf die Zielgerade ein und wird finalisiert. Bis dahin muss diese Funktion sowohl beim Kompilieren als auch Starten mit `--enable-preview` explizit eingeschaltet werden. Und natürlich sollte man sie noch nicht produktiv einsetzen, da sich potenziell immer noch Aspekte ändern können.

*Listing 2* zeigt nochmal den Funktionsumfang mit einem Beispiel aus einer früheren JEP-Dokumentation. Der Selektionsausdruck o wird auf verschiedene Typen geprüft. Je nachdem, ob es sich um einen String, eine Farbe, einen Punkt oder ein `int`-Array handelt, wird der jeweilige `case`-Zweig aufgerufen. In einem normalen Switch darf im `case` nur gegen primitive Zahlen beziehungsweise deren Wrapper sowie Strings und Enums verglichen werden. Hier wird dagegen geschaut, ob o einem beliebigen Datentyp (inklusive Array) entspricht. Ist das der Fall, wird automatisch ein Cast auf diesen Datentyp ausgeführt und der Wert in einer Variablen gespeichert. Diese kann dann direkt weiterverarbeitet werden. Sollte o eine Null-Referenz sein, dann wird der optionale Zweig `case null` aufgerufen und nicht, wie bei Switch-Statements sonst üblich, eine NullPointerException geworfen. Ein früher notwendiger, expliziter Null-Check vor dem Switch kann somit entfallen.

Die Type Patterns kann man mit den `when`-Clauses (bisher Guarded Patterns) dann noch verfeinern, wie in *Listing 1* gezeigt. Damit wirkt der Switch aufgeräumter, ist leichter zu lesen und man spart geschachtelte `if-else`-Abfragen innerhalb der `case`-Zweige.

Zum Pattern Matching for Switch gibt es noch einiges mehr zu erzählen. Besonders hervorzuheben ist die bessere Typsicherheit bei der Verwendung von Sealed Classes durch die Vollständigkeitsprüfung (Exhaustiveness) des Compilers. Da hat sich das Verhalten in Java 19 allerdings nicht verändert. Eine detaillierte Beschreibung dazu findet ihr in meinem Artikel in der kommenden Ausgabe 1/23 der Java aktuell.

## Record Patterns zum Destrukturieren von Daten

Dieser neue Pattern-Typ ist erstmals im JDK 19 dabei und wurde eigentlich schon länger erwartet. Bisher konnten neben den Constant-Patterns (Zahlen, Strings, Enums) im Switch nur die Type-Patterns (instanceof) verwendet werden. Mit den Record Patterns kommt jetzt ein zweiter wichtiger Aspekt des Pattern Matching hinzu, die Dekonstruktion der passenden Datenstrukturen. Die Definition in der Scala-Dokumentation *[5]* beschreibt das sehr treffend:

*Pattern matching is a mechanism for checking a value against a pattern. A successful match can also deconstruct a value into its constituent parts. It is a more powerful version of the switch statement in Java and it can likewise be used in place of a series of if/else statements.*

Der letzte Satz nimmt noch einmal Bezug auf das, was wir mit den Switch Expressions und dem Pattern Matching for Switch schon heute in Java verfügbar haben.

Records sind transparente, unveränderbare Träger für Daten, ähnlich zu Tuple-Datentypen, aber zusätzlich mit einem Namen versehen *[6]*. Sie wurden im OpenJDK 14 als Preview eingeführt und mit 16 finalisiert. Mit ihnen können gemeinsam mit Sealed Classes algebraische Datentypen gebildet werden und genau dafür werden sie beim Pattern Matching benötigt. Auch hierauf werden wir im nachfolgenden Artikel genauer schauen. Records sind aufgrund ihrer kompakten Syntax und der Immutability auch einfach zum Modellieren beliebiger Datenklassen als Alternative zu POJOs/JavaBeans einsetzbar.

Der JEP 405 für Record Patterns war ursprünglich schon für das JDK 17 eingeplant, damals noch in der Kombination mit Array Patterns. Dabei sind jedoch noch Schwierigkeiten aufgetreten und so hat man diese beiden Pattern-Typen mittlerweile getrennt. Wann die Array Patterns folgen, ist aktuell unklar. Außerdem werden noch weitere Pattern-Typen folgen, wie beispielsweise POJO-Patterns oder die Dekonstruktion von Datentypen über Factory-Methoden.

Records sind effektiv nur eine spezielle Art der Klassen-Definition, bei der den Entwicklern durch den Compiler das Schreiben von unnötigem Ballast abgenommen wird. Dadurch lassen sich sehr schlank reine Datentypen als Objekte modellieren und daraus Instanzen anlegen. Beim Record Pattern gehen wir nun den umgekehrten Weg und dekonstruieren beziehungsweise zerlegen eine Datenstruktur in ihre Einzelteile. Das würde man auch mit dem Type Pattern und instanceof erreichen. Da müssen sich Entwickler jedoch zusätzlich durch explizite Aufrufe der Getter-/Accessor-Methoden die relevanten Informationen nach dem Matchen des Musters manuell auslesen. Das Record Pattern stellt nun eine deklarative Beschreibung des Musters dar, wobei die Extraktion in die Bestandteile unter der Haube passiert. Die Entwickler können direkt mit den relevanten, bereits zugewiesenen Informationen weiterarbeiten. Da spart Codezeilen und macht den Sourcecode konsistenter, besser lesbar und damit auch wartbarer.

In *Listing 3* werden die zwei Datenklassen Point und Rectangle definiert. Beim Prüfen auf den Typ kann man direkt die Properties des Records mit Variablennamen versehen (ähnlich einem Konstruktoraufruf). Bei einem Treffer werden diesen Variablen die Inhalte des Records (Properties) zugewiesen und man kann sie direkt verwenden.

Das funktioniert auch mit geschachtelten („nested") Records. Zum Beispiel suchen wir deklarativ nach dem Muster eines Rechtecks, das zwei Punkte enthält. Mit den Koordinaten der Punkte kann nach der automatischen Zerlegung dann weitergearbeitet werden. Zusätzlich darf man sich auch eine Referenz auf das Rechteck oder den Punkt selbst in einer weiteren Variablen merken. Hierbei lassen sich verschiedene Granularitätsstufen mischen. Wenn man sich also nur für die Koordinaten des ersten Punktes interessiert, kann der zweite Punkt quasi „anonym" bleiben. In Zukunft wird es da irgendwann auch ein Platzhalter-Pattern (beispielsweise mit dem Unterstrich _) für nicht benötigte Referenzen geben, so wie das Scala oder andere funktionale Sprachen beim Pattern Matching bereits anbieten.

Weitere Pattern-Typen sind bereits in der Planung. Wir dürfen gespannt sein, wann Array-, POJO- und Wildcard-Patterns sowie das Pattern Matching über Factory-Methoden bereitstehen.

## Nebenläufigkeit reloaded

Bereits seit 2018 wird im Rahmen des Project Loom an der Einführung der sogenannten virtuellen Threads (zwischendurch Fibers genannt) gearbeitet. Im JDK 19 haben sie es jetzt erstmals als Preview geschafft. Wenn es gut läuft, könnten sie theoretisch nach einer zweiten Preview in Java 20 bereits mit dem LTS-Release OpenJDK 21 im Herbst 2023 finalisiert werden. Diese Version würde dann von der Wichtigkeit her in einer Reihe mit Java 5 (Generics), Java 8 (Lambda, Stream API) und Java 9 (JPMS) stehen. Ob es allerdings so kommt, lässt sich im Moment noch nicht sagen.

Im Wiki des Projekts Loom werden die Virtual Threads folgendermaßen angekündigt: *JVM features and APIs for supporting easy-to-use, high-throughput, lightweight concurrency and new programming models.*

Nebenläufigkeit richtig zu implementieren, war nie leicht. Aber bereits seit den Anfängen in den 90er Jahren bietet Java diverse Möglichkeiten, Anwendungscode zu parallelisieren und konkurrierend auszuführen. Im Laufe der fast drei Jahrzehnte kamen laufend weitere Möglichkeiten hinzu. Das Ziel ist, lang laufende Berechnungen möglichst einfach und effizient auf mehrere, gleichzeitig laufende Threads zu verteilen und somit zu beschleunigen.

```
record Point(int x, int y) {
}
record Rectangle(Point p1, Point p2) {
}

Object mayBeAPoint = new Point(0, 0);
Object mayBeARectangle = new Rectangle(new Point(0, 1), new Point(10, 6));

if (mayBeAPoint instanceof Point ( int x, int y)){
System.out.println(x + y);
}

if (mayBeARectangle instanceof Rectangle(Point(int x1, int y1), Point(var x2, var y2)) r) {
int area = Math.abs(x2 - x1) * Math.abs(y2 - y1);
System.out.println(String.format("Fläche des Rechtecks (%s): %s", r, area));
}

if (mayBeARectangle instanceof Rectangle(Point(int x1, int y1), Point p2)) {
System.out.println(String.format("Koordinaten des P1 vom Rechteck: (%s; %s)", x1, y1));
}
```

*Listing 3*

Die maximale Anzahl gleichzeitiger Threads ist jedoch oft der Flaschenhals. Bei Webanwendungen wird beispielsweise für jeden eingehenden Request ein separater Programmfaden benötigt. Bisher entsprach ein Java-Thread immer einem Betriebssystem-Thread. Diese sind sehr ressourcenhungrig und damit in der Anzahl stark begrenzt. Der Versuch, nur ein paar Tausend Threads gleichzeitig zu starten, endet schnell in einem Out-of-Memory-Error. Werden die Threads alternativ über einen Pool in der Anzahl begrenzt, antwortet die Anwendung aufgrund vieler blockierender Operationen nicht mehr wie gewünscht. Arbeiten also sehr viele Nutzer gleichzeitig mit dem Websystem, kommt es schon bei einer eigentlich noch überschaubaren Anzahl an seine Grenzen. Insbesondere wenn die Bearbeitung einzelner Threads länger dauert, weil auf blockierende Datenstrukturen oder externe Dienste wie Datenbanken gewartet werden muss.

Reaktive Programmierung kann bei diesen Problemen nützlich sein, erhöht aber die Komplexität und bringt neue Herausforderungen bezüglich Wart- und Debugbarkeit mit sich. Außerdem müssen alle Glieder in der Kette für die reaktive Programmierung vorbereitet sein, beispielsweise braucht es für die Datenbankzugriffe die entsprechenden Treiber.

## Mit virtuellen Threads wird alles besser

Virtuelle Threads erlauben es nun, die konkurrierende Verarbeitung parallel ausgeführter Aufgaben auf eine beliebige (sehr große) Anzahl an Threads zu verteilen. Dabei lässt sich nun sogar leicht les- und gut wartbarer Code schreiben. Den kann man zudem, wie herkömmlichen sequenziellen Code, mit den Standardmitteln einfach debuggen. Virtuelle verhalten sich dabei aus Sicht des Programmierers wie normale Threads, werden aber nicht 1:1 auf die Betriebssystem-Threads gemappt. Stattdessen gibt es einen Pool von Träger-Threads (Carrier Threads), denen virtuelle Threads kurzzeitig zugewiesen werden. Sobald der virtuelle Thread auf eine blockierende Operation stößt und wartet, wird er vom Träger entkoppelt. Dieser kann dann einen anderen virtuellen Thread (einen neuen oder einen zuvor blockierten) übernehmen. Dadurch lassen sich jetzt plötzlich Millionen paralleler Threads in einer JVM verwalten.

```
Thread.Builder threadBuilder = createThreadBuilder();
threadBuilder.start(() -> {
  // im Thread auszuführender Code …
  System.out.println(Thread.currentThread().isVirtual());
});

…

private static Thread.Builder createThreadBuilder() {
return Thread.ofVirtual();
// return Thread.ofPlatform();
}
```

*Listing 4*

Die virtuellen Threads wurden transparent in die bestehende Klassenhierarchie des JDK integriert. Die neuen Factory-Methoden erstellen je nach Typ sogenannte ThreadBuilder, die unter der Haube entweder eine Instanz der Klasse java.lang.Thread (Plattform-Threads) beziehungsweise java.lang.VirtualThread erzeugen *(siehe Listing 4)*.

Über die ThreadBuilder können noch Anpassungen an der Konfiguration vorgenommen werden. Durch diese saubere Abstraktion/Trennung lässt sich innerhalb bestehender Frameworks sehr einfach auf die virtuelle Thread-Variante wechseln. So profitieren selbst bestehende Anwendungen ohne Änderungen am Code durch den Wechsel auf Java 19 oder höher. Solange sich die Virtual Threads allerdings noch im Preview-Modus befinden, sollte man mit dem Einsatz in produktiver Software vorsichtig sein.

Blockierende Operationen halten den ausführenden Thread nun nicht mehr unnötig auf. Mit nur einem kleinen Pool schwergewichtiger Carrier Threads kann somit sehr effizient eine hohe Zahl von Requests parallel verarbeitet werden. Dabei wird auch nicht mehr die Gesamtperformance durch den Overhead bei den Plattform-Threads heruntergezogen. *Abbildung 1* zeigt das Ergebnis eines kleinen, nicht ganz repräsentativen Performance-Experiments.

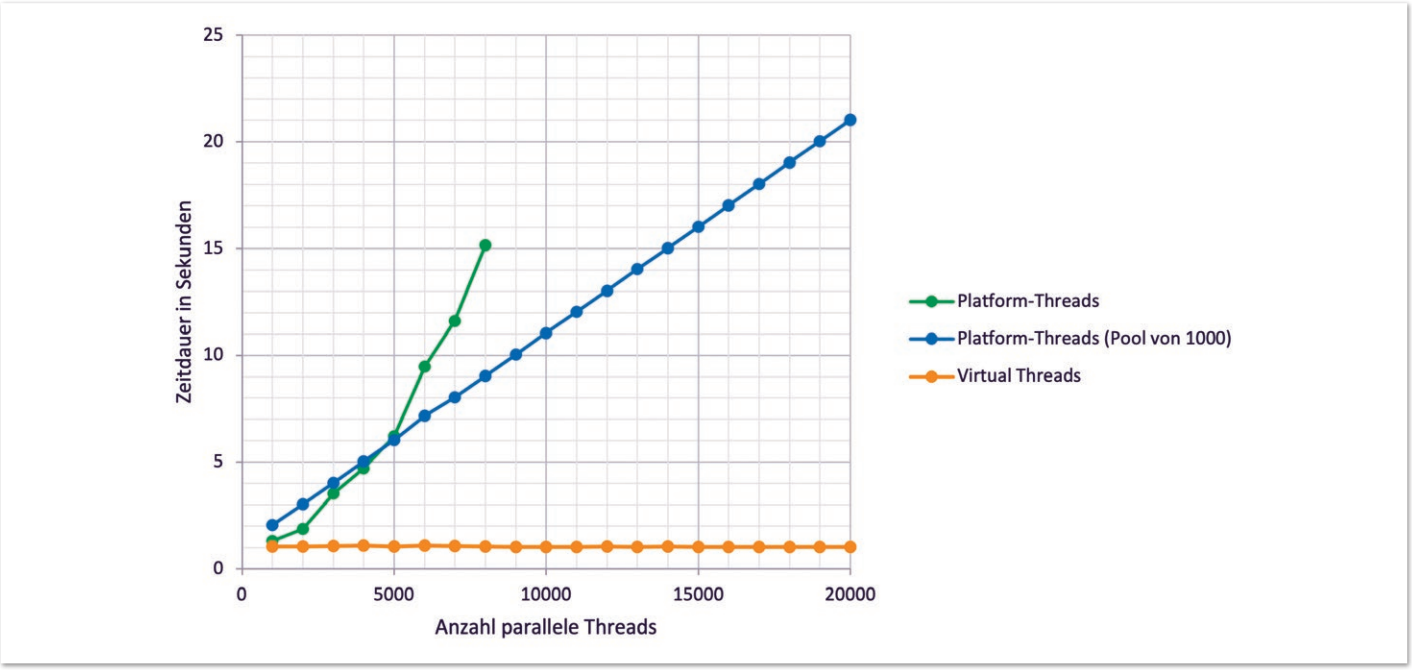



*Abbildung 1*

```
[77.854s][warning][os,thread] Failed to start thread "Unknown thread" - pthread_create failed (EAGAIN) for attrib-
utes: stacksize: 1024k, guardsize: 4k, detached.
[77.855s][warning][os,thread] Failed to start the native thread for java.lang.Thread "pool-10-thread-8150"
Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or
process/resource limits reached
```

*Listing 5*

```
private static final ExecutorService executor = Executors.newCachedThreadPool();

Future<String> task1 = executor.submit(() -> { … }
Future<String> task2 = executor.submit(() -> { … }
Future<String> task3 = executor.submit(() -> { … }

System.out.println(task1.get());
System.out.println(task2.get());
System.out.println(task3.get());
```

*Listing 6*

Es werden in Tausenderschritten nacheinander Plattform- und virtuelle Threads über einen ExecutorService ausgeführt. Wobei die jeweiligen Threads jeweils eine Sekunde warten, um den Zugriff auf eine externe Schnittstelle zu simulieren. Am Ende wird die Gesamtzeit gemessen. Während bei den virtuellen Threads (`Executors.newVirtualThreadPerTaskExecutor()`) sich ein nahezu konstantes Ergebnis zeigt (gelbe Linie, die durchschnittliche Gesamtlaufzeit ist 1,05 Sekunden), steigt die Zeit bei den Plattform-Threads (`Executors.newFixedThreadPool(numberOfThreads)`) exponentiell an (grüne Linie). Zudem ist auf meinem Rechner mit den Default-VM-Einstellungen bereits bei 9.000 parallelen Threads Schluss, wie die Fehlermeldung in *Listing 5* zeigt.

Bei einer Begrenzung auf 1.000 gleichzeitige Plattform-Threads (`Executors.newFixedThreadPool(1000)`) bricht die Anwendung zwar nicht ab, zeigt aber immer noch einen linearen Anstieg (rote Linie) und damit einen signifikanten Overhead gegenüber den virtuellen Threads.

Virtuelle Threads werden die konkurrierende Programmierung revolutionieren. Sie ermöglichen das Schreiben von gut verständlichem Code, der jedoch scheinbar beliebig skaliert. Ressourcenhungrige Betriebssystem-Threads werden nicht unnötig blockiert, wenn auf Locks, blockierende Datenstrukturen, Antworten vom Dateisystem oder externen Services gewartet werden muss. Bis typische Backend-Frameworks wie Spring virtuelle Threads unterstützen, wird noch etwas Zeit vergehen. Doch dann profitieren eben auch Altanwendungen, da sie ohne Änderungen am Code unter der Haube die virtuellen Threads automatisch verwenden können.

```
try (var scope = new StructuredTaskScope.ShutdownOn-
Failure()) {
  Future<String> task1 = scope.fork(() -> { … }
Future<String> task2 = scope.fork(() -> { … }
Future<String> task3 = scope.fork(() -> { … }

  scope.join();
scope.throwIfFailed();

  System.out.println(task1.get());
System.out.println(task2.get());
System.out.println(task3.get());
}
```

*Listing 7*

## Structured Concurrency

Virtuelle Threads sind die Voraussetzung für ein weiteres spannendes Feature. Besteht eine Aufgabe aus verschiedenen Teilen, die parallel verarbeitet werden sollen (etwa Zugriff aufs Filesystem, Lesen aus der Datenbank und Aufrufe von Fremdsystemen), so setzt man typischerweise ExecutorServices ein. In *Listing 6* werden drei Aufgaben gleichzeitig ausgeführt.

Die Anwendung blockiert, bis die Ergebnisse aller drei Tasks fertig sind und gibt dann die Ergebnisse aus. Doch was passiert, wenn bei einem der Tasks ein Fehler auftritt? Wie kann man dann die anderen Tasks abbrechen? Wie kann man allgemein die Tasks abbrechen, wenn die gesamte Berechnung nicht mehr benötigt wird? Wenn Task 1 sehr lange läuft, bekommt man erst sehr spät mit, dass Task 2 oder 3 Fehler produziert haben. Natürlich lassen sich für diese Probleme Lösungen entwickeln, die sind aber komplex und häufig schlecht lesbar. Auch das Debuggen wäre nicht einfach, da man in den Thread-Dumps die Tasks nicht den jeweiligen Threads aus dem Pool zuordnen kann.

Bei der Structured Concurrency ersetzen wir den `ExecutorService` durch einen `StucturedTaskScope`, bei dem man verschiedene Strategien (hier `ShutdownOnFailure`) auswählen kann. Wie in *Listing 7* zu sehen, bekommt man durch dieses API bessere und viel lesbarere Optionen, um auch mit Fehlersituationen umzugehen.

Mit scope.join() wird gewartet, bis alle Tasks erfolgreich erledigt sind. Schlägt einer fehl oder wird abgebrochen, werden auch die anderen beiden direkt beendet. Mit throwIfFailed() kann der Fehler außerdem weitergegeben werden und das Ausgeben der Ergebnisse wird übersprungen.

Dieser neue Ansatz bringt einige Vorteile. Zum einen bilden Task und Sub-Tasks im Code eine abgeschlossene, zusammengehörige Einheit. Die Threads kommen nicht mehr aus einem Thread-Pool mit schwer-

```
$ javac --enable-preview -source 19 --add-modules jdk.incubator.concurrent StructuredConcurrencyDemo.java
$ java --enable-preview --add-modules jdk.incubator.concurrent StructuredConcurrencyDemo
```

*Listing 8*

gewichtigen Plattform-Threads, stattdessen wird jede Unteraufgabe in einem neuen virtuellen Thread ausgeführt. Bei Fehlern werden noch laufende Sub-Tasks abgebrochen. Zudem sind bei Fehlersituationen die Informationen besser, weil die Aufrufhierarchie sowohl in der Code-Struktur als auch in der Exception sichtbar ist.

Da die Structured Concurrency zunächst als Incubator Feature ausgeliefert wird, benötigt es beim Kompilieren und Ausführen einige zusätzliche Informationen *(siehe Listing 8)*. Nicht vergessen, es könnte in den nächsten Java Releases noch einige, auch grundlegende Änderung geben.

## Fazit und Ausblick

Im Vergleich zu den früheren, halbjährlichen Releases erscheint der Umfang des OpenJDK 19 mit „nur" 7 umgesetzten JEPs sehr überschaubar. Insbesondere wenn man bedenkt, dass bereits im September 2023 das nächste LTS-Release (Java 21) ansteht. Aber Oracle bleibt seiner Strategie treu und nimmt nur das in die Releases auf, was bis zu einem bestimmten Stichtag fertig ist. Dadurch schaffen sie es, den festen Zeitplan von zwei Releases pro Jahr (immer im März und September) einzuhalten.

Es macht ja auch nicht die Masse, sondern die Klasse. Denn bei genauerem Hinschauen fällt auf, dass auch im JDK 19 mit den Virtual Threads und der Structured Concurrency sehr wegweisende Themen aufgenommen und viele bereits länger in Bearbeitung befindliche wieder ein Stück weiterentwickelt wurden. Gerade das Pattern Matching wird uns noch über einige Jahre begleiten. Die Grundlagen mit den Switch Expressions, Sealed Classes, Records und Type Patterns wurden bis zum JDK 17 gelegt. Vielleicht können im nächsten LTS-Release JDK 21 auch schon das Pattern Matching for Switch und die Record Patterns finalisiert werden. Doch da gibt es noch viele weitere Ideen, die in diesem Umfeld folgen werden.

Leider standen zum Zeitpunkt der Entstehung dieses Artikels noch keine Inhalte für das nächste Release (OpenJDK 20) fest. Hier lohnt aber immer mal wieder ein Blick auf die Projektseite *[7]*. Denn schon Ende 2022 müssen die umzusetzenden Features/JEPs feststehen, damit das Release bis zur Veröffentlichung im März 2023 stabilisiert werden kann. Die Spannung steigt, ob dann für das LTS-Release 21 zunächst die in Arbeit befindlichen Themen beendet werden oder ob uns Oracle parallel mit weiteren Neuerungen überraschen wird. Die Value Objects und Primitive Classes aus dem Inkubator-Projekt Valhalla sind solche Features, die schon von vielen sehnsüchtig im JDK erwartet werden.

Das Java-Ökosystem ist lebendiger denn je und weiterhin sehr innovativ. Konkurrenz wie beispielsweise Python (hauptsächlich aufgrund von Machine/Deep Learning), JavaScript, Go und die C-basierten Sprachen beleben das Geschäft. Java, das besonders im Unternehmensanwendungsumfeld vertreten ist, wird aber weiterhin ein gehöriges Wort mitreden.

Eine spannende Frage ist, wie viele Java-Projekte überhaupt schon die modernen Java-Versionen (11+) verwenden. Laut diversen, nicht repräsentativen Umfragen ist Java 8 immer noch stark vertreten. Gerade der Schritt von 8 auf 11 kann aufwendig sein, davor scheuen sich viele Entwicklungsabteilungen. Aus der Erfahrung heraus fallen die Updates danach leichter. Die geringste Hürde bei den Update-Strategien stellen halbjährliche Aktualisierungen auf die jeweils aktuelle Major-Version dar. Durch die kurze Zeitspanne zwischen den Long-Term-Support-Versionen von nur noch zwei Jahren stellen Versionssprünge zum nächsten LTS-Release jedoch auch keine hohe Hürde mehr dar. Das Hauptproblem ist dann vermutlich eher die noch fehlende Unterstützung von Bibliotheks-, Framework- und Tool-Herstellern. Für die stellen die kurzen Release-Zyklen tatsächlich eine große Herausforderung dar.

So wie es aussieht, wird uns als Java-Entwickler so schnell nicht langweilig werden. Die Zukunftsaussichten für die Sprache und das Ökosystem sind weiterhin sehr rosig.

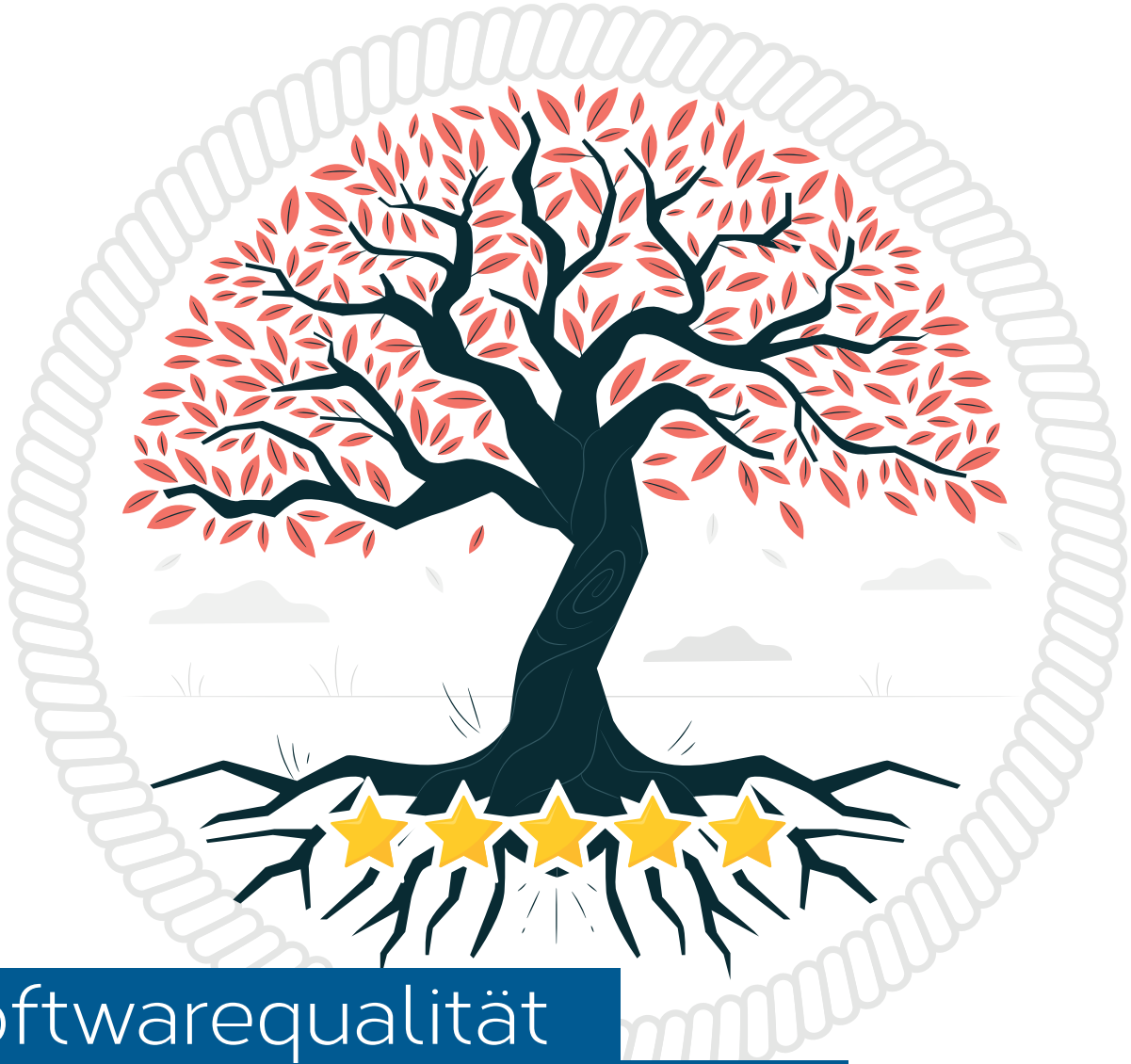
## Referenzen:


[1] *https://openjdk.java.net/projects/jdk/19/*
[2] *https://jdk.java.net/19/release-notes*
[3] *https://javaalmanac.io/jdk/19/apidiff/18/*
[4] *https://openjdk.java.net/projects/amber/*
[5] *https://docs.scala-lang.org/tour/pattern-matching.html]*
[6] *https://jax.de/blog/datenklassen-in-java-einfuehrung-in-java-records/*
[7] *https://openjdk.org/projects/jdk/20/*



**Falk Sippach**
embarc Software Consulting GmbH
*falk@jug-da.de*


Falk Sippach ist bei der embarc Software Consulting GmbH als Softwarearchitekt, Berater und Trainer stets auf der Suche nach dem Funken Leidenschaft, den er bei seinen Teilnehmern, Kunden und Kollegen entfachen kann. Bereits seit über 15 Jahren unterstützt er in meist agilen Softwareentwicklungsprojekten im Java-Umfeld. Als aktiver Bestandteil der Community (Mitorganisator der JUG Darmstadt) teilt er zudem sein Wissen gern in Artikeln, Blog-Beiträgen sowie bei Vorträgen auf Konferenzen oder User-Group-Treffen und unterstützt bei der Organisation diverser Fachveranstaltungen. Falk twittert unter @sippsack.



# Softwarequalität an der Wurzel gepackt

**Yves Schubert, iteratec GmbH**

*Ein Sofa kann gemütlich weich sein, ein Apfel appetitlich rot, ein Lautsprecher einen wohligen Klang haben oder ein Fahrrad leicht und aus einem Guss geformt sein. Wir ordnen allen Dingen mehr oder weniger subjektiv empfundene Eigenschaften zu und folgern aus Emotionen und Messwerten bewusst oder unbewusst, dass ein Produkt als Ganzes eine bestimmte Qualität aufweist. Meist ist der Begriff „Qualität" eher positiv belegt, ist aber streng genommen vorerst wertneutral.*

*Auch in der Softwareentwicklung sprechen wir von „Qualität" als die Beschaffenheit einer Software. Hier tauchen einige Fragen auf: Was ist das eigentlich, „Softwarequalität"? Sprechen wir hier vom Herstellungsprozess? Sprechen wir über das Laufzeitverhalten der Software? Das Aussehen? Wie sie sich „anfühlt"?*

*Sie ahnen es schon: Wir haben es mit einer extrem facettenreichen Zusammenstellung unterschiedlicher Blickrichtungen und Ebenen zu tun, wenn wir die Qualität einer Software messen wollen. In diesem Artikel versuche ich zu ergründen, wie aus diesen vielen Facetten eine gute Softwarequalität entstehen kann. Nehmen Sie am besten auf einem gemütlichen Sofa Platz und lesen Sie weiter.*

Lassen Sie mich mit einer Frage beginnen: Was ist eigentlich Qualität? Jeder hat eine intuitive Vorstellung von Qualität. Wenn man tiefer einsteigt, gelangt man schnell in den psychoanalytischen Bereich, bei dem es um die Marke eines Produkts geht oder darum, dass dem Produkt eine höhere Qualität zugesprochen wird, wenn es viele Menschen benutzen oder viele (positiv) davon sprechen. Eine große Rolle spielen hier also Emotionen, die zunächst durch Werbung transportiert und auch durch eigene Erfahrungen (etwa durch Haptik) generiert werden. Durch das gezielte Erschaffen dieser positiven Emotionen werden auch weitere positive Eigenschaften suggeriert. Selbst eher negativ auffallende Merkmale können dadurch neutralisiert werden. Genau wie diese schwer fassbaren subjektiven Merkmale kann eine Qualität auch durch harte Fakten und Zahlen dargestellt werden. So kann etwa durch Angabe einer objektiv messbaren Messtoleranz auf die Qualität eines Produkts geschlossen werden.

Obwohl sich die Herstellung von Software in seinem Gesamtprozess grundlegend von der eines anfassbaren Produkts unterscheidet, trifft man hier auf ähnliche Erkenntnisse. Eine Software kann sich in der Verwendung „gut anfühlen" oder eine bestimmte Performance-Messzahl erfüllen. Es kann aber auch sein, dass Performance oder die Bedienbarkeit keine Rolle spielen, wenn es darum geht, nächtliche Batch-Jobs zuverlässig zu verrichten.

## Was also verstehen wir unter Softwarequalität?

Eine Definition, angelehnt an ISO/IEC 25010: „Softwarequalität ist die Summe aller subjektiven und objektiven Eigenschaften, die dazu beitragen, die Anforderungen an eine Software zu erfüllen."

Der Kern ist hier, dass nur eine bestimmte Menge von Eigenschaften ausgewählt wird. Diese müssen dazu beitragen, bestimmte Anforderungen zu erfüllen. Umgekehrt: Eine Eigenschaft, die keinen Einfluss auf eine vorausgesetzte Anforderung hat, trägt auch nicht zur Softwarequalität bei.

Die verwendbaren Eigenschaften sind in dem Qualitätsmodell der bereits genannten Norm ISO/IEC 25010 dargestellt und in der Regel allen Softwareentwicklern und Softwareentwicklerinnen bekannt oder zumindest intuitiv präsent. Diese sind: Funktionalität, Benutzbarkeit, Zuverlässigkeit, Effizienz, Änderbarkeit, Übertragbarkeit, Sicherheit und Kompatibilität. Auf die Details werde ich hier nicht eingehen, denn das ist hier nicht das Thema. Vielmehr möchte ich darüber reden, in welchen Ebenen des Softwareentwicklungszyklus Einfluss auf diese Qualitätsziele genommen wird. Fangen wir also von vorne an.

## Am Anfang steht die Idee

Schon die erste Idee beeinflusst die Qualität des späteren Produkts maßgeblich. Hier werden bereits erste Ziele und Anforderungen formuliert. Je klarer hier schon Qualitätsziele definiert werden und je klarer die Systemgrenzen gezogen werden, desto zuträglicher ist das am Ende für die Produktqualität. Denn alle an dem Projekt Beteiligten können und werden sich in ihrer täglichen Arbeit an diesen Zielen orientieren. Und das ist der eigentliche Knackpunkt: Fehlen diese Leitplanken, führt das dazu, dass die Projektbeteiligten im Laufe des Entwicklungszyklus ihren auf Erfahrung basierenden eigenen Maßstab anlegen und im schlimmsten Fall gar nicht geforderte oder gar sich konkurrierende Qualitätsziele implementieren. Meist geschieht das nicht explizit, sondern implizit; die Folge davon ist eine zu komplexe oder zu unflexible Architektur der Software und des Entwicklungsprozesses.

## Weiter voran mit dem Vorgehensmodell

In der Natur der Sache liegt es, dass das Vorgehensmodell mal mehr, mal weniger frei bestimmt werden kann. Das hängt zusammen mit der Historie der Firma, einem vorhandenen Qualitätsmanagement und nicht zuletzt mit der Konstellation der Menschen, die das Projekt angehen. Es ergibt immer Sinn zu hinterfragen, wie man voranschreiten möchte. Ein Vorgehensmodell darf auf keinen Fall zum Selbstzweck angewendet werden, sondern muss im Einklang mit den Zielen des jeweiligen Projekts und des Unternehmens sein. Ein gegebenenfalls vorhandenes Qualitätsmanagement muss den Rahmen schaffen, sodass die Qualitätsziele eingehalten werden können. Denn diese sind das Maß der Dinge und bestimmen letztendlich auch den wirtschaftlichen Erfolg des Projekts. Sie sollten maßgeblich dafür sein, welches Modell gewählt wird. Was hier so einfach steht, ist in der Realität eine sehr schwierige und vor allem andauernde Tätigkeit, deren Herausforderung darin besteht, die richtige Dosis von Kommunikationskanälen, Dokumenten, Vorschriften und Richtlinien zu finden. Historisch haben wir hier eine enorme Bandbreite von Modellen, von Wasserfall und SPICE bis zu Scrum und XP. Ein Vorgehensmodell zu verstehen und richtig anzuwenden allein ist schon eine große Kunst. Ich möchte hier nicht diskutieren, welches Modell „besser" ist, denn allen Modellen, ob schwergewichtig oder nicht, muss fairerweise zugeschrieben werden, dass sie existieren, weil sie echte Schmerzen lösen wollen. Dabei gibt es sicherlich Vorgehensmodelle, die das Erlangen bestimmter Qualitätsziele begünstigen oder behindern. Schon in sehr frühen Konferenzen über Softwareengineering, wie der NATO Software Engineering Conference 1968 *[1]*, wurde viel über dieses Thema gesprochen *(siehe Abbildung 1)*.

*Abbildung 1: Ein Foto von der NATO-Software-Engineering-Konferenz 1968 (© Robert McClure und Brian Randell, http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html)*

Hier wurde nichts Geringeres als das Wort „Softwarekrise" geprägt. Denn in den 1950er Jahren liefen die meisten größeren Softwareprojekte aus dem Ruder. Sie benötigten ein zu hohes Budget, wurden selten in gegebener Zeit fertig und die Qualität der Software wurde sehr gering eingeschätzt. Ein Zitat zum Thema Softwaredesign von Douglas Ross möchte ich hier nicht unerwähnt lassen:

> **The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from.**
>
> **Douglas Ross**
> *NATO Software Engineering Conference, 1968*

Dieser Satz zielt darauf ab, dass es gefährlich ist zu versuchen, eine Software von vornherein komplett zu spezifizieren und sie dann auch tatsächlich genauso zu implementieren. Weiter heißt es: „The projects that are called successful, have met their specifications. But those specifications were based upon the designers' ignorance before they started the job."

Schon damals war bekannt, dass die Softwarewelt eine sehr dynamische Welt ist. Dennoch wurden zu Beginn eines Projekts, an dem eigentlich sehr viel von dem zu implementierenden System noch unbekannt ist, wichtige Entscheidungen getroffen. Was aber hat das mit Qualitätszielen zu tun? Sehr viel, denn die hohen Kosten, die in dem Fall auftreten, dass erst sehr spät im Entwicklungszyklus Fehlentscheidungen erkannt werden, können mit sorgfältig gewählten Qualitätszielen kompensiert werden. Früh formulierte Qualitätsziele können, wenn sie im Projekt richtig gelebt werden, die Weichen in die richtige Richtung stellen.

## Das Team spricht mit

Wo wir schon bei gelebten Zielen sind: Die Menschen in dem Projektumfeld sind also diejenigen, die die Qualitätsziele leben und die Verantwortung für die Einhaltung der Qualitätsziele tragen, aber auch die Früchte davon ernten. Was viele vergessen: In jedem Vorgehensmodell bestehen gewisse Freiheiten, die vom Team ausgenutzt werden können und auch müssen. Alle Regeln sollen eine Hilfestellung sein, eine Checkliste. Die Menschen sind diejenigen, die kommunizieren und es in der Hand haben, wie kommuniziert wird. Sie sind diejenigen, die entscheiden, wie umfangreich bestimmte Dokumente sind, wie Regeln ausgelegt werden. Entscheidend sind also in diesem Sinne die Schnittstellen zwischen den Projektbeteiligten. Sind diese Schnittstellen transparent, werden Qualitätsziele intrinsisch eingehalten und Projekte im entsprechenden Rahmen erfolgreich. Etwas gewagt gesagt: Der Erfolg spricht dann für sich.

Eine Qualitätssicherungsabteilung kann nicht in der Lage sein, nur durch Tests eine gute Qualität herbeizuführen. Denn durch Testen wird nicht aufgedeckt, ob ein minimaler Code geschrieben wurde, um Anforderungen abzudecken. Es wird auch nicht erkannt, ob Code lesbar geschrieben oder einfach zu verändern ist. Es ist unmöglich, durch Tests herauszufinden, ob Code unnötig komplex ist. Ebenso wird nicht geprüft, ob die Systemarchitektur für die Lösung passend ist *[2]*. Die Message ist: Silodenken ist unangebracht. Jeder Beteiligte ist für das Erreichen der Qualitätsziele in vollem Maße erforderlich. Transparente und einfache Kommunikationswege sind essenziell dafür.

## Geschichten erzählen

Ein sehr schönes Mittel zur Kommunikation von Qualitätszielen sind die sogenannten Qualitätsszenarien, die auch in arc42, einer beliebten Vorlage für Architekturdokumentation, fester Bestandteil des Kapitels 10.2 sind *[3]*. Qualitätsszenarien sind Formulierungen von Szenarien, die auf bestimmte Qualitätsziele (etwa aus ISO/IEC 25010) einzahlen und dabei einen konkreten Sachverhalt darstellen. So sind sie ein Hilfsmittel zur Kommunikation mit Kunden, Stakeholdern sowie Entwicklerinnen und Entwicklern und können dabei Einfluss auf die Priorisierung der Anforderungen und damit auch auf die technische Architektur haben. Qualitätsszenarien können außerdem gut getestet werden, weil sie messbare Metriken liefern. Wir haben hier also ein Werkzeug, das uns den Stoff liefert, den wir für den konsistenten Transport der Qualitätsziele über alle Ebenen hinweg so dringend benötigen.

## Die vier Schlüsselmetriken

Das Team rund um Google Cloud und die Autoren des sehr empfehlenswerten Buches „Accelerate" *[4]* haben in dem wissenschaftlichen Projekt namens „DORA" (DevOps Research and Assessments) aus den erfassten Daten vier Metriken herausgearbeitet, die Aufschluss über die Performance und Stabilität eines Projekts geben sollen. Das ist zum einen die Auslieferungsfrequenz („Deployment Frequency"), also die Häufigkeit, wie oft ein Projekt in Produktion gebracht werden kann. Außerdem wird die Zeit gemessen, wie lange es dauert, bis ein Code-Commit in Produktion ist („Change Lead Time"). Eine weitere Metrik ist die Zeit, die für das Wiederherstellen eines Dienstes gebraucht wird, wenn dieser von einem Fehler betroffen ist („Mean Time To Restore", MTTR). Die vierte Messzahl ist der prozentuale Anteil von Ausfällen oder Fehlern, die nach einer Auslieferung auftreten („Change Fail Percentage"). Mit dem sogenannten DORA DevOps Quick Check *[5]* werden genau diese vier Metriken erhoben, anschließend wird eine Einschätzung dazu gegeben, wo man mit seinem Projekt steht *(siehe Abbildung 2)*.

Das Spannende an diesen Metriken ist die Tatsache, dass sie auf den ersten Blick auf Schnelligkeit aus sind und sich womöglich die Frage aufdrängt: „Warum sollten wir so schnell ausliefern müssen? In unserem Prozess ist das sowieso nicht vorgesehen und die Kunden brauchen nicht mehrmals am Tag eine neue Version!" Bei genauerer Überlegung kommt man jedoch zu dem Schluss, dass die Erreichung einer hohen Ausliefergeschwindigkeit kein Selbstzweck ist, sondern die hier erhobenen Zahlen Rückschlüsse auf die Softwarequalität des Produkts zulassen. Denn gute Zahlen werden hier nur erreicht, wenn die Prozesse flüssig laufen, die Kommunikation funktioniert und der Code während der Paketierung sehr gut getestet wird – wenn insgesamt also sehr auf möglichst viel Automatisierung gesetzt wird. Mittel zum Zweck sind hier der konsequente Einsatz einer langen Reihe von „Fähigkeiten", die in einem umfangreichen Katalog zusammengestellt sind *[6]*. Wie bereits erwähnt, sind es eben genau die vier „Key Metrics", die auf eine gute oder schlechte Softwarequalität schließen lassen. Und, das ist das Bemerkenswerte, zwar eben nicht nur auf quantitativ messbare Merkmale, sondern auch auf schwierig messbare Merkmale wie Architektur oder Teamzusammenhalt.

## Fazit

Was ist Qualität? Was ist Softwarequalität? Diese Fragen sind und bleiben schwer zu beantworten. Eines kann man festhalten: Es liegt dann eine gute Qualität vor, wenn die Qualitätsziele so gut wie möglich erfüllt sind. Anders kann man auch sagen, eine gute Qualität liegt dann vor, wenn möglichst viele am Projekt beteiligten Menschen zufrieden sind. Qualität ist also auch vom Betrachter abhängig und teils auch subjektiv.

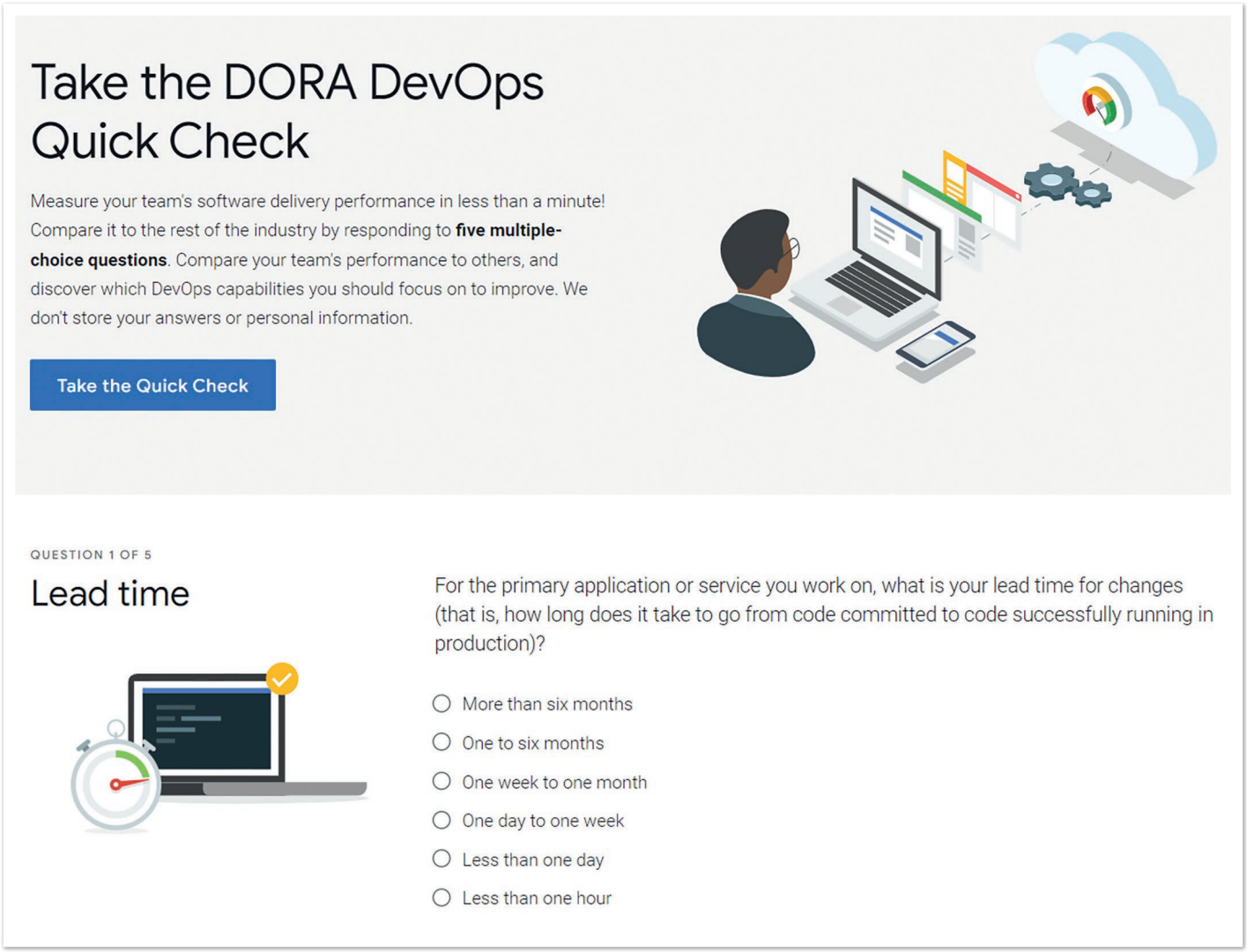



*Abbildung 2: Screenshot des DORA DevOps Quick Check [5] [© Google Cloud]*

Wenn man sich die Dialoge der NATO Software Engineering Conference 1968 genauer ansieht, erkennt man, dass die heutigen Probleme scheinbar immer noch die gleichen sind. Der Unterschied ist aber, dass wir inzwischen einige Lösungen vorliegen haben und eine Menge Arbeit geleistet wurde, um aus der Softwarekrise der 1950er Jahre Lehren zu ziehen. Im Laufe der Zeit entstanden viele Herangehensweisen und Modelle, die helfen, eine hohe Softwarequalität zu erreichen. Es liegt aber in unser aller Verantwortung, diese Lösungen und Werkzeuge auch sinnvoll einzusetzen. Wir müssen also von unserem bequemen Sofa aufstehen und dafür sorgen, dass wir Qualitätsziele einfordern oder definieren und an alle Projektbeteiligten in geeigneter Weise kommunizieren.

## Quellen


[1] NATO Software Engineering Conference (1968), *http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html*

[2] David Farley (2021): Modern Software Engineering: Doing What Works to Build Better Software Faster, Addison-Wesley Professional

[3] arc42 Template, Kapitel 10, *https://docs.arc42.org/section-10/*

[4] Ph.D. Forsgren, Nicole, Jez Humble, Gene Kim (2018): Accelerate: The Science Behind Devops: Building and Scaling High Performing Technology Organizations, IT Revolution Press

[5] DORA Quickcheck, *https://www.devops-research.com/quick-check.html*

[6] DORA Capability catalog, *https://www.devops-research.com/research.html#capabilities*



**Yves Schubert**
Iteratec GmbH
*yves.schubert@iteratec.com*


Seit über 15 Jahren bin ich beruflich und auch privat in vielen Softwareprojekten unterschiedlichster Technologiebereiche maßgeblich beteiligt. Von neuen Technologien lasse ich mich immer begeistern und probiere daher, möglichst viel von dieser schnelllebigen Softwarewelt mitzunehmen. Neue Trends klopfe ich dabei aber auf ihre Nachhaltigkeit hin ab und wäge kritisch ab, ob und wie diese eingesetzt werden können. In der Praxis wende ich die Prinzipien des Software Craftsmanship an.

# Mutation Testing – Wer testet die Tests?

**Nicolai Mainiero, sidion GmbH**

*Unit- und Integrationstests sind ein essenzieller Bestandteil, um qualitativ hochwertige Software entwickeln zu können. Sie helfen nicht nur sicherzustellen, dass die Software alle notwendigen Anforderungen erfüllt, sondern unterstützen auch bei Wartung, Refactoring und Weiterentwicklung, indem sie dafür sorgen, dass keine neuen Fehler in die Software eingebracht werden. Doch wie kann die Qualität der Tests gewährleistet werden?*

Beim Testen in der Softwareentwicklung bedient man sich unter anderem zweier dynamischer Testverfahren: Black-Box-Testverfahren *[1]* und White-Box-Testverfahren *[2]*. Beide dienen zum Auffinden von Fehlern, indem das Testobjekt ausgeführt und das tatsächliche Ergebnis gegen das erwartete Ergebnis geprüft wird. Beim Black-Box-Testverfahren werden die Testfälle auf Basis der Spezifikation entworfen. Um die Anzahl der Testfälle überschaubar zu halten, werden diese systematisch reduziert, indem nur Grenzwerte betrachtet werden, Äquivalenzklassen gebildet werden, Entscheidungstabellen zum Einsatz kommen oder Use-Case-bezogen getestet wird. Diese Reduzierung erfordert Erfahrung beim Testen und es kann leicht passieren, dass zum Beispiel eine Äquivalenzklasse vergessen wird oder sich durch eine spätere Änderung der Spezifikation die Äquivalenzklassen verändern. Ebenfalls ist es sehr aufwendig, anhand von Black-Box-Tests zu zeigen, dass eine hinreichende Testabdeckung erreicht worden ist. Im Gegensatz dazu kann beim White-Box-Testverfahren die Testabdeckung (Code Coverage) in verschiedenen Metriken relativ einfach gemessen werden. Da auf die innere Funktionsweise zugegriffen werden kann, können beispielsweise folgende Qualitätskriterien ermittelt werden:

- Zeilenüberdeckung: Ausführung aller Quellcode-Zeilen
- Anweisungsüberdeckung beziehungsweise Knotenüberdeckung: Ausführung aller Anweisungen
- Zweigüberdeckung beziehungsweise Kantenüberdeckung: Durchlaufen aller möglichen Kanten von Verzweigungen des Kontrollflusses
- Bedingungsüberdeckung beziehungsweise Termüberdeckung (mehrere Varianten): Durchlaufen aller möglichen ausschlaggebenden Belegungen bei logischen Ausdrücken in Bedingungen
- Pfadüberdeckung (mehrere Varianten): Betrachtung der Pfade durch ein Modul

Diese Metriken können mit den gängigen Werkzeugen wie JaCoCo, OpenClover oder JCov automatisch ermittelt werden.

## Testabdeckung

Eine hohe prozentuale Testabdeckung sollte zu einem großen Vertrauen in die Tests führen, dass diese zum Beispiel alle Fallunterscheidungen abtesten. Leider sagt eine 90-prozentige Testabdeckung noch nichts über die Qualität der Tests aus. Der Unittest in *Listing 1* für die `biggerThanTen`-Methode aus *Listing 2* erreicht zwar 100 % Abdeckung der Kanten und sogar der Zeilen, der Grenzfall 10 wird jedoch vom Unittest nicht berücksichtigt und das Fehlverhalten wird nicht erkannt.

```
@Test
public boolean test_biggerThanTen() {
  assertTrue(biggerThanTen(11));
  assertFalse(biggerThanTen(9));
}
```

*Listing 1: Unittest für die biggerThanTen-Methode*

```
public boolean biggerThanTen(int x) {
  if(x >= 10) {
     return true;
  } else {
     return false;
  }
}
```

*Listing 2: Die `biggerThanTen`-Methode*

Die Testabdeckung suggeriert hier Sicherheit, die aber aufgrund des mangelhaften Tests nicht vorhanden ist. Ein weiteres Beispiel, wie es durchaus in Legacy Code zu finden wäre, zeigt *Listing 3*.

```
@Test
public boolean test_biggerThanTenV2() {
  biggerThanTen(11);
  biggerThanTen(10);
  biggerThanTen(9);
}
```

*Listing 3: Pathologischer Unittest für die biggerThanTen-Methode*

In diesem Fall werden alle Äquivalenzklassen in dem Unittest berücksichtigt und die Testabdeckung würde wieder 100 % melden, allerdings wurde vergessen, den Rückgabewert der Funktion zu bestätigen. Zugegeben, diesen Fehler sollte bereits die statische Codeanalyse aufdecken und verhindern.

## Mutation Testing

Um qualitativ gute Software zu schreiben, werden auch gute Tests für die Software benötigt; sich auf Testabdeckung zu verlassen, ist kein Garant für aussagekräftige Test. Welche Möglichkeiten gibt es, Softwareentwicklern ein Werkzeug an die Hand zu geben, das sie dabei unterstützt, gute Tests für die von ihnen entwickelte Software zu schreiben? Mutation Testing ist eine Möglichkeit, Entwicklern zu helfen, bessere Tests zu schreiben.

| Ursprüngliche Bedingung | Mutierte Bedingung |
|---|---|
| < | <= |
| <= | < |
| > | >= |
| >= | > |

*Tabelle 1: Mögliche Mutationen bei Vergleichen*

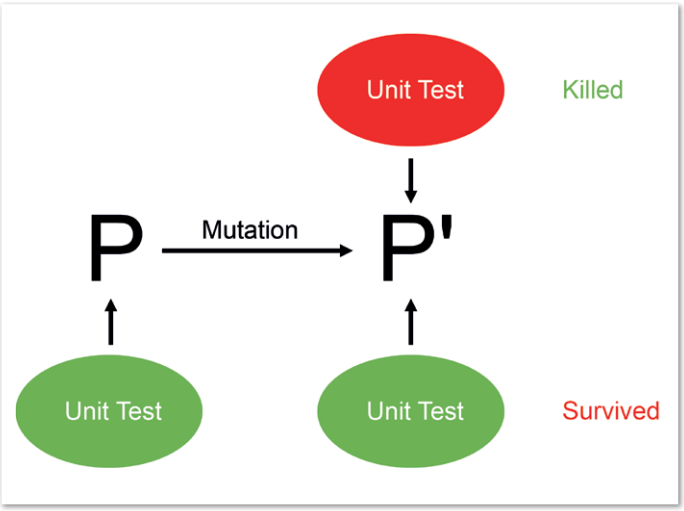



*Abbildung 1: Vorgehen des Pittest (© Nicolai Mainiero)*

```
public class Calculator {
  public int add(int a, int b) { return a + b; }
  public int subtract(int a, int b) { return a - b; }
  public int multiply(int a, int b) { return a * b; }
  public int divide(int a, int b) { return b == 0 ? 0 : a / b; }
}
```

*Listing 4: Ein einfacher Taschenrechner*

Mutation Testing ist keine neue Methode, die erste Veröffentlichung zu dieser Methode stammt aus dem Jahr 1978: „Hints on Test Data Selection: Help for the Practicing Programmer" *[3]*. Eine erste Implementierung für FORTRAN erschien im Jahr 1980. Mit Certitude wurde die Methode auf das Design von ASICs ausgedehnt, um die Entwurfsprüfung unabhängiger von den Fähigkeiten des Chipdesigners zu machen. Mittlerweile gibt es für die meisten gängigen Programmiersprachen geeignete Bibliotheken *[4] [5] [6]*, um diese Methode in den Entwicklungsprozess zu integrieren.

Mutation Testing basiert auf zwei grundlegenden Prinzipien:

- Kompetente Programmierer erstellen Programme, die nahezu korrekt sind
- Der Kopplungseffekt: Testdaten, die alle Programme unterscheiden, die sich von einem korrekten Programm nur durch einfache Fehler unterscheiden, sind so empfindlich, dass sie auch komplexere Fehler unterscheiden

Das bedeutet, dass durch Einbringen kleiner Fehler in den Quellcode, der getestet werden soll, überprüft werden kann, wie robust die Tests für diese Einheit sind. Dieser Ansatz ist ähnlich zu dem Chaos Engineering *[7]*, das zum Beispiel Netflix mit seinem Chaos Monkey populär gemacht hat. Indem zufällig Services oder Infrastruktur-Komponenten deaktiviert werden, wird überprüft, wie robust das System gegen Störungen ist.

## Mutante

Anhand von Pittest *[4]* soll die Funktionsweise von Mutation Testing erläutert werden. Zunächst werden mithilfe sogenannter Mutation Operators (oder Mutators) Änderungen in den bereits übersetzten Bytecode der Einheit, die getestet werden soll, eingebracht. *Tabelle 1* zeigt, welche Mutationen bei Vergleichen durchgeführt werden.

```
class CalculatorTest {

  private Calculator calculator = new Calculator();

     @ParameterizedTest
     @CsvSource({"0,0,0"})
     void givenTwoNumbers_when_add_then_sum(int a, int b, int expectedResult) {
           // given
           // when
           int actualResult = calculator.add(a, b);
           // then
           assertThat(actualResult).isEqualTo(expectedResult);
     }

     @ParameterizedTest
     @CsvSource({"1,-1,2", "1,0,1", "1,1,0"})
     void givenTwoNumbers_when_subtract_then_difference(int a, int b, int expectedResult) {
           // given
           // when
           int actualResult = calculator.subtract(a, b);
           // then
           //assertThat(actualResult).isEqualTo(expectedResult);
     }

     @ParameterizedTest
     @CsvSource({"1,1,1", "2,3,6"})
     void givenTwoNumbers_when_multiply_then_product(int a, int b, int expectedResult) {
           // given
           // when
           int actualResult = calculator.multiply(a, b);
           // then
           assertThat(actualResult).isNotNegative();
     }

     @ParameterizedTest
     @CsvSource({"1,1,1"})
     void givenTwoNumbers_when_divide_then_quotient(int a, int b, int expectedResult) {
           // given
           // when
           int actualResult = calculator.divide(a, b);
           // then
           assertThat(actualResult).isEqualTo(expectedResult);
     }
}
```

*Listing 5: Unittest der Calculator-Klasse*

Pittest bringt über 20 Gruppen solcher Mutationen mit, die in eine „*Default*, *Stronger* und *All*"-Gruppe zusammengefasst worden sind und darüber aktiviert werden können. Neben den bereits erwähnten Vergleichen werden beispielsweise auch die Increment- (i++) und Decrement-Operation (i--) vertauscht, Rechenoperatoren vertauscht, Vergleiche negiert, Rückgabewerte auf Konstanten reduziert und vieles mehr. Aus jedem Programm P, für das es einen Unittest gibt, erzeugt Pittest 1 bis n Mutanten, indem es die Mutation Operators auf das Programm P anwendet und damit ein P' erzeugt, wie in *Abbildung 1* dargestellt.

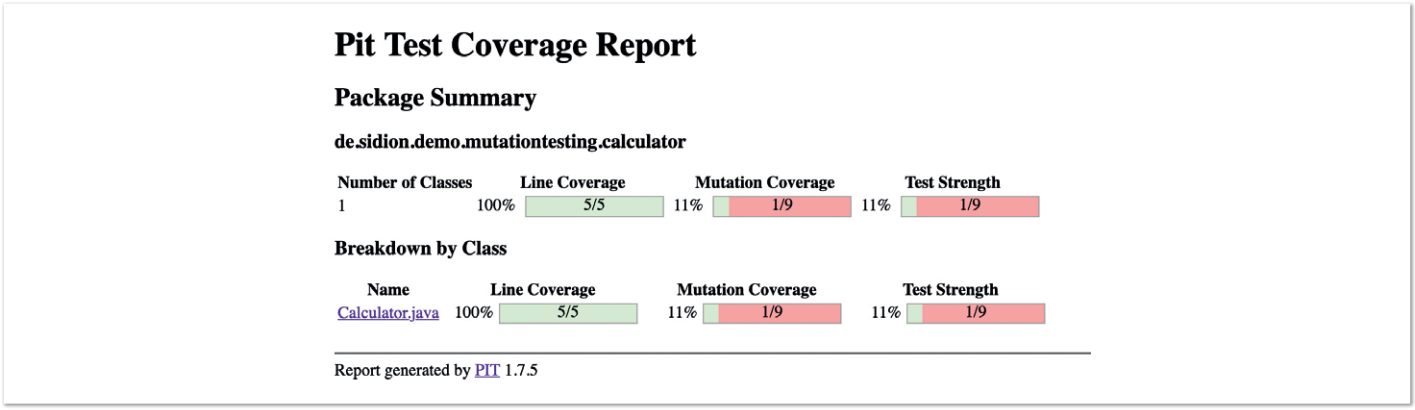
Für jedes mutierte Programm P' werden die relevanten Unittests ausgeführt und überprüft, ob der Test noch erfolgreich ist oder nicht. Ist der Test nach wie vor positiv, hat die Mutation überlebt. Schlägt er fehl, wurde die Mutation erfolgreich getötet. Ziel ist es, keine überlebende Mutation zu erhalten.

## Praxis

Am sehr einfachen Beispiel eines Taschenrechners, der die vier Grundrechenarten beherrscht, soll gezeigt werden, wie Mutation Testing den Entwickler beim Schreiben besserer Tests unterstützen kann *(siehe Listing 4)*.

Der Entwickler hat für die Klasse folgenden parametrisierten Unittest geschrieben *(siehe Listing 5)*. Leicht zu erkennen ist, dass die Tests entweder fehlerhafte Implementierungen haben oder die Parameter nicht aussagekräftig genug sind, um alle Äquivalenzklassen abzudecken.

Dennoch erreicht dieser Unittest, wenn er ausgeführt wird, eine 100-prozentige Testabdeckung. Das wird aus dem Pittest Coverage Report in *Abbildung 2* erkenntlich. Trotz 100 % Line Coverage wird von den neun Mutationen, die Pittest erzeugt, nur eine erkannt und



**Pit Test Coverage Report**

**Package Summary**

**de.sidion.demo.mutationtesting.calculator**

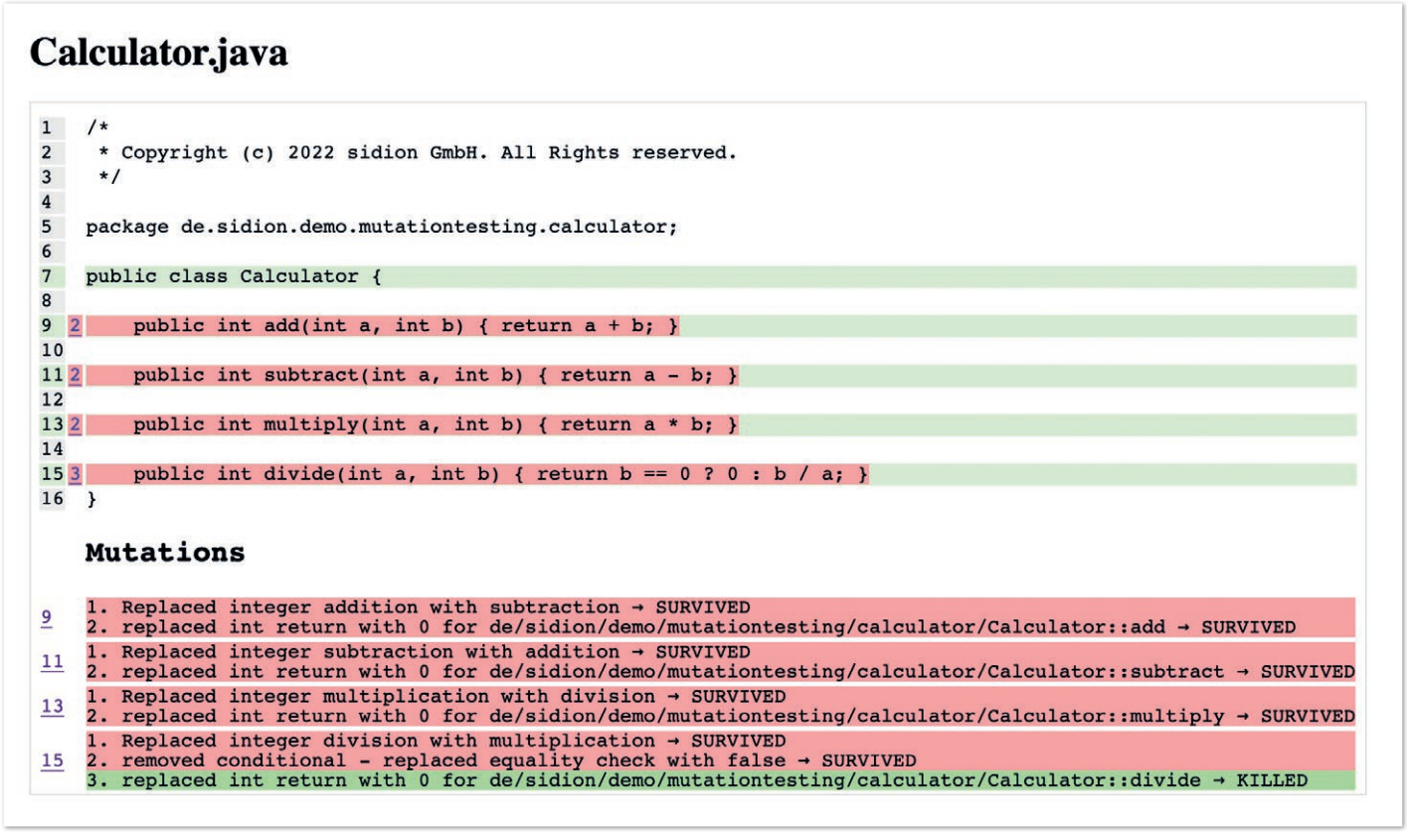
| Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| 1 | 100% | 5/5 | 11% | 1/9 | 11% | 1/9 |

**Breakdown by Class**

| Name | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| Calculator.java | 100% | 5/5 | 11% | 1/9 | 11% | 1/9 |

Report generated by PIT 1.7.5

*Abbildung 2: Pit Test Coverage Report – schlechte Mutation Coverage*



**Calculator.java**

```
1   /*
2    * Copyright (c) 2022 sidion GmbH. All Rights reserved.
3    */
4
5   package de.sidion.demo.mutationtesting.calculator;
6
7   public class Calculator {
8
9  2    public int add(int a, int b) { return a + b; }
10
11 2    public int subtract(int a, int b) { return a - b; }
12
13 2    public int multiply(int a, int b) { return a * b; }
14
15 3    public int divide(int a, int b) { return b == 0 ? 0 : b / a; }
16  }
```

**Mutations**

```
9   1. Replaced integer addition with subtraction → SURVIVED
    2. replaced int return with 0 for de/sidion/demo/mutationtesting/calculator/Calculator::add → SURVIVED
11  1. Replaced integer subtraction with addition → SURVIVED
    2. replaced int return with 0 for de/sidion/demo/mutationtesting/calculator/Calculator::subtract → SURVIVED
13  1. Replaced integer multiplication with division → SURVIVED
    2. replaced int return with 0 for de/sidion/demo/mutationtesting/calculator/Calculator::multiply → SURVIVED
15  1. Replaced integer division with multiplication → SURVIVED
    2. removed conditional - replaced equality check with false → SURVIVED
    3. replaced int return with 0 for de/sidion/demo/mutationtesting/calculator/Calculator::divide → KILLED
```

*Abbildung 3: Detaillierter Test-Report*

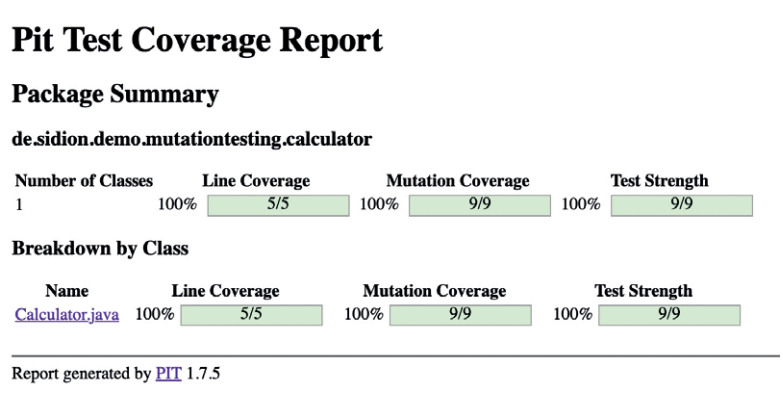
```
class CalculatorTest {

	private Calculator calculator = new Calculator();

	@ParameterizedTest
	@CsvSource({"0,0,0", "-1,1,0", "1,2,3"})
	void givenTwoNumbers_when_add_then_sum(int a, int b, int expectedResult) {
		// given
		// when
		int actualResult = calculator.add(a, b);
		// then
		assertThat(actualResult).isEqualTo(expectedResult);
	}

	@ParameterizedTest
	@CsvSource({"1,-1,2", "1,0,1", "1,1,0"})
	void givenTwoNumbers_when_subtract_then_difference(int a, int b, int expectedResult) {
		// given
		// when
		int actualResult = calculator.subtract(a, b);
		// then
		assertThat(actualResult).isEqualTo(expectedResult);
	}

	@ParameterizedTest
	@CsvSource({"1,1,1", "2,3,6"})
	void givenTwoNumbers_when_multiply_then_product(int a, int b, int expectedResult) {
		// given
		// when
		int actualResult = calculator.multiply(a, b);
		// then
		assertThat(actualResult).isEqualTo(expectedResult);
	}

	@ParameterizedTest
	@CsvSource({"1,1,1", "1,0,0", "6,2,3"})
	void givenTwoNumbers_when_divide_then_quotient(int a, int b, int expectedResult) {
		// given
		// when
		int actualResult = calculator.divide(a, b);
		// then
		assertThat(actualResult).isEqualTo(expectedResult);
	}
}
```

*Listing 6: Verbesserter Unittest der Calculator-Klasse*

erfolgreich getötet. Die anderen acht führen nach wie vor zu erfolgreich ausgeführten Unittests.

Die detaillierte Auswertung des Reports auf Klassenebene, zu sehen in *Abbildung 3*, markiert alle Zeilen, in denen mindestens eine Mutation überlebt hat. Weiter unten wird genau ausgeführt, welche Mutation durchgeführt wurde und dennoch zu einem erfolgreichen Unittest geführt hat. Diese Informationen helfen nun dabei, bessere Unittests *(siehe Listing 6)* für diesen Taschenrechner zu schreiben.

Wird Pittest mit den Anpassungen an den Unittest erneut ausgeführt, erhält man eine Mutation Coverage und Test Strength von 100 %, wie in *Abbildung 4* zu sehen. Wichtig ist, dass es nicht immer möglich ist, diese Werte auf 100 % zu bekommen. Es kann sein, dass ein Unittest vollständig und korrekt ist, aber der Mutation Test dennoch einen Fehler anzeigt. Wird normalerweise keine Zeilenüberdeckung



**Pit Test Coverage Report**

**Package Summary**

**de.sidion.demo.mutationtesting.calculator**

| Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| 1 | 100% | 5/5 | 100% | 9/9 | 100% | 9/9 |

**Breakdown by Class**

| Name | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| Calculator.java | 100% | 5/5 | 100% | 9/9 | 100% | 9/9 |

Report generated by PIT 1.7.5

*Abbildung 4: Pit Test Coverage Report – sehr gute Mutation Coverage*

von 100 % angestrebt, wird normalerweise auch keine Mutation Coverage und Test Strength von 100 % anvisiert.

## Fazit

Wie gesehen, ist Testabdeckung allein keine aussagekräftige Metrik für die Qualität vorhandener Unittests. Werden diese jedoch durch Mutationstests ergänzt, kann die Qualität der Unittests auf einfache Weise, auch von unerfahrenen Softwareentwicklern, verbessert werden. Es werden unvollständige und ineffektive Tests, toter oder redundanter Code und Bugs gefunden. Mutation Testing härtet die Testsuite, macht sie robuster und gibt zusätzliches Vertrauen bei größeren Refactorings. Das initiale Setup ist relativ einfach, es ist aber wichtig, alle Beteiligten aufzuklären und zu informieren. Insgesamt ist es ein nützliches Tool, das bei der Qualitätssicherung von Unittests hilfreich ist. Der Quellcode für die Beispiele ist unter *[8]* zu finden.

## Quellen


[1] Wikipedia – Die freie Enzyklopädie, „Black-Box-Test," [Online]. Available: *https://de.wikipedia.org/wiki/Black-Box-Test*.

[2] Wikipedia – Die freie Enzyklopädie, „White-Box-Test," [Online]. Available: *https://de.wikipedia.org/wiki/White-Box-Test*.

[3] R. A. Demillo und F. Sayward, „Hints on Test Data Selection: Help for the Practicing Programmer," Computer, Bd. 11, Nr. 4, pp. 34 - 41, 1978.

[4] „Pit Test," [Online]. Available: *https://pitest.org/*.

[5] „Stryker Mutator," [Online]. Available: *https://stryker-mutator.io/*.

[6] „Ruby Mutant," [Online]. Available: *https://github.com/coreyjs/ruby-mutant*.

[7] Wikipedia – Die freie Enzyklopädie, „Chaos engineering," [Online]. Available: *https://en.wikipedia.org/wiki/Chaos_engineering*.

[8] „Mutation Testing Beispiel," [Online]. Available: *https://gitlab.com/sidion/vortraege/2022/javaforumstuttgart/mutationtesting*.

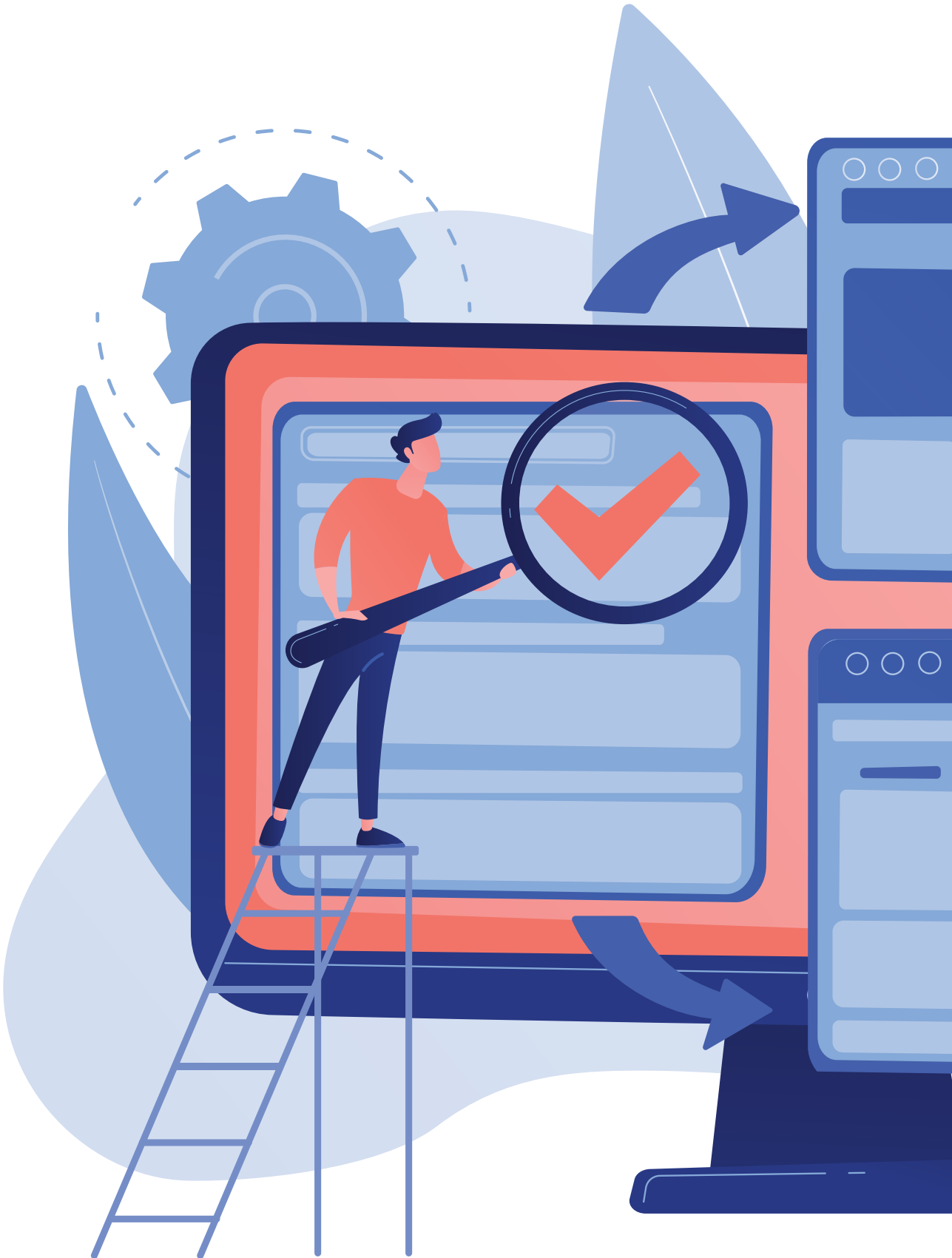


**Nicolai Mainiero**
sidion GmbH
*nicolai.mainiero@sidion.de*


Nicolai Mainiero ist Diplom-Informatiker und arbeitet als Software Developer bei der sidion GmbH. Er entwickelt seit über 14 Jahren Geschäftsanwendungen in Java, Kotlin und Clojure für unterschiedlichste Kundenprojekte. Dabei setzt er vor allem auf agile Methoden wie Kanban. Außerdem interessiert er sich für funktionale Programmierung, Microservices und reaktive Anwendungen.

# Advanced ArchUnit: Tests auf Bytecode-Analysen aufbauen

Thomas Ruhroth, msg systems, Kai Schmidt



*ArchUnit ist ein beliebtes Tool für die Sicherstellung von Architekturvorgaben für JVM-Sprachen. Da die Prüfung im Rahmen von Unittests erfolgt, ist sie für Entwickler einfach und schnell in die lokale Testumgebung und damit in eine Continuous-Integration-Pipeline einzubauen. Grundlegende Regeln, wie Zugriffsberechtigungen zwischen Paketen, lassen sich leicht prüfen. Ebenso liefert ArchUnit konfigurierbare Standard-Architekturen, mit denen man beispielsweise eine hexagonale Architektur sicherstellen kann. Schwieriger wird es, wenn man Prüfungen umsetzen möchte, die (noch) nicht in ArchUnit vorgesehen sind oder die von den bestehenden abweichen.*
*In diesem Artikel erleben wir einen Ausflug über die verschiedenen Ebenen des ArchUnit-API, um zu verstehen, welche Arten von Prüfungen auf welcher Ebene durchgeführt werden und wie diese Ebenen miteinander kommunizieren.*

## Einleitung/Motivation

ArchUnit *[1]* ist die bevorzugte Methode der Autoren, in einem Projekt definierte Architektur-Eigenschaften sicherzustellen. Im Gegensatz zu anderen Tools sind die Tests im Code hinterlegt und über Unittests ausführbar. (Eine Einführung ist in *[2]* zu finden.)

Durch viele vordefinierte Prüfungen und deren Fluent-API bietet es einen leichten Einstieg, gute und weitgreifende Architekturtests zu erstellen. So lassen sich eigene Prüfbibliotheken bereitstellen oder Prüfungen durchführen, die einen direkten Zugriff auf das von ArchUnit bereitgestellte Bytecode-Modell benötigen.

Ein Beispiel ist die Prüfung eines im Projekt eingesetzten Architekturmusters namens „Functional Core/Imperative Shell" *[3]*. Das Muster bringt Prinzipien aus der funktionalen Welt in die Architektur objektorientierter/imperativer Sprachen. Da es in ArchUnit derzeit für dieses Muster keine vordefinierten Tests und keine Möglichkeit gibt, dieses durch das derzeitige Lang-API sicherzustellen, war es unser Einstieg in die tiefere Programmierung von Conditions und Libraries für ArchUnit. Das Vorhaben erwies sich als kompliziert, ist daher nicht als Einführungsbeispiel geeignet und würde den Rahmen dieses Artikels sprengen.

Die Dokumentation deckt die Möglichkeiten von ArchUnit nicht vollständig ab – häufig hilft in diesen Fällen ein Blick in den Sourcecode. Dieser Artikel soll den Einstieg in die Erstellung eigener Prüfungen und eigener ArchUnit-Bibliotheken erleichtern. Als Implementierungsbeispiel dient die Verhinderung von als deprecated markierten Aufrufen. Um zuvor ein Grundverständnis für die Architektur von ArchUnit aufzubauen, startet der Artikel mit einem kurzen Überblick.

## ArchUnit – Prüfen von Architektureigenschaften

Eine Besonderheit von ArchUnit ist, dass die Analysen auf Bytecode-Ebene durchgeführt werden. Dies bietet den Vorteil, dass alle JVM-Sprachen und Prüfungen nach einem Bytecode-Enhancement unterstützt werden. Auf der anderen Seite bedeutet das, dass Eigenschaften, die sich nicht im Bytecode widerspiegeln, nicht geprüft werden können – beispielsweise generische Definitionen oder Annotationen mit der `RetentionPolicy.SOURCE`.

ArchUnit arbeitet auf drei verschiedenen Abstraktionsebenen *(siehe Abbildung 1)*. Auf der konkretesten Ebene, dem Core-API, hat man den vollen Zugriff auf das Bytecode-Modell und kann Strukturen auf ihre Eigenschaften abfragen. Beispielsweise lassen sich für eine Methode die darin aufgerufenen Methoden ermitteln. Darauf aufbauend zeichnet sich die zweite Abstraktionsebene namens Lang-API aufgrund seines Fluent-Designs mit einer guten Verständlichkeit des mit ihm geschriebenen Codes aus. Ebenso ist dies in der dritten Abstraktionsebene (Library-API) der Fall. Hiermit können vordefinierte Architekturen wie die Onion-Architektur oder Abgleiche mit PlantUML-Modellen definiert werden. Übergreifende Regeln, die beispielsweise zyklische Abhängigkeiten verbieten, befinden sich ebenfalls auf dieser Ebene.

Da das Lang-API unter ArchUnit-Anwendern bekannt sein sollte, führt uns der Überblick zunächst zu dieser mittleren Ebene.

## Lang-API

Das Lang-API bietet über eine Art Baukastensystem die Möglichkeit, verständliche Regeln häufig auftretender Prüfungen, wie zum Beispiel von Paketabhängigkeiten, zu erstellen. Eine Regel bildet sich nach einem definierten Schema: `$STRUCTURE that $PREDICATE should $CONDITION`

STRUCTURES können verschiedene Strukturelemente von Java sein, dazu gehören `classes()`, `methods()`, `members()`, `fields()`, `codeUnits()` und `constructors()`. Die STRUCTURE stellt den Einstiegspunkt dar, über den definiert wird, auf welchen Strukturelementen die Prüfung durchgeführt werden soll. Die hierfür zu berücksichtigenden Klassen bauen sich entweder über die Annotation @AnalyzeClasses und die Angabe der Java-Pakete oder im Testcode über einen `ClassFileImporter` auf.

Die Menge von Elementen wird über das PREDICATE auf eine Untermenge gefiltert, auf der die über die Condition definierte Regel geprüft wird. Diese formale Struktur bildet sich direkt in den Umsetzungen ab (siehe *Listing 1* Zeile a). In diesem Fall verwenden wir als STRUCTURE die häufig verwendete Definition `classes()`, um Prüfungen auf Klassenebene auszuführen. Für viele Anwendungsfälle stehen solche Prädikate bereits zur Verfügung. Beispielsweise selektiert die Methode `resideInAPackage` in unserer Verwendung alle Klassen, die in einem übergeordneten Paket „source" enthalten sind – beispielsweise über die Package-Deklaration `com.source.api` oder `de.javaaktuell.source`.

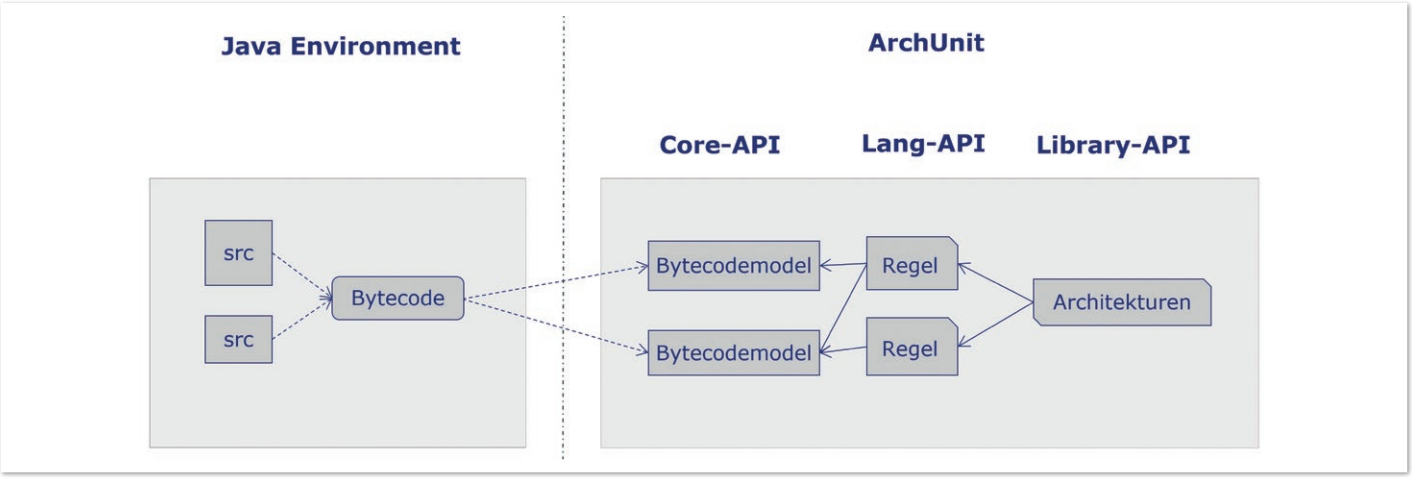



*Abbildung 1: Abstraktionsebenen von ArchUnit (Quelle Thomas Ruhroth)*

```
a) classes().that().resideInAPackage("..source..")
b) .should().dependOnClassesThat().resideInAPack-
age("..foo..")
```

*Listing 1: Beispiel einer ArchUnit Regel im Lang-API*

Im zweiten Schritt werden über die CONDITION die Prüfungen definiert, die auf diese Elemente angewendet werden sollen. Auch hierfür stellt ArchUnit vorgefertigte Methoden bereit, die CONDITIONs werden mit der Methode `should` eingeleitet *(siehe Listing 1 Zeile b)*. Über die so zusammengestellte Regel wird definiert, dass jede einzelne Klasse aus `source` von mindestens einer Klasse mit einem übergeordneten Paket `foo` abhängig sein muss.

## Library-API

Über die vorgestellte Regel ließen sich die Schichtenzugriffe beliebiger Architekturen definieren. Für gängige Regeln und Architekturen sind derartige Prüfungen bereits auf einer höheren Abstraktionsebene, dem Library-API, gebündelt. Zum Sicherstellen einer Schichtenarchitektur muss konfiguriert werden, wo sich die Schichten befinden und welche Zugriffsmöglichkeiten auf andere Schichten zulässig sind. Da die Zugriffsmöglichkeiten der verschiedenen Schichten für eine hexagonale/Onion-Architektur in ArchUnit bereits vordefiniert sind, genügt in diesem Fall die Zuordnung der Pakete zu den vorgegebenen Schichten *(siehe Listing 2)*.

Während das Lang-API und existierende Architekturprüfungen im Library-API gut dokumentiert sind, findet man zu den Möglichkeiten, die sich nah am Bytecode abspielen, wenig Quellen. Auch im Rahmen dieses Artikels kann keine detaillierte Beschreibung dieses grundlegenden APIs erfolgen. Dennoch soll ein Grundverständnis geschaffen werden, damit das Erforschen des API und weiteres Tüfteln leichter fällt.

```
onionArchitecture()
  .domainModels("architecture.domain.model..")
  .domainServices("architecture.domain.service..")
  .applicationServices("architecture.application..")
  .adapter("persistence", "architecture.adapter.db..")
  // eventuell weitere adapter
```

*Listing 2: Beispielverwendung der Onion-Architektur in ArchUnit*

## Core-API

Den technischen Unterbau für das Lang-API bildet das Core-API, das mit einer Abstraktion des JVM-Bytecodes die Grundlage für alle Prüfungen darstellt. Das Core-API deckt dabei alle Konstrukte ab, die über das Bytecode-Modell repräsentiert sind. Somit ergibt sich eine Repräsentation der Programmstruktur als Graph, auf dem sich die Prüfungen aufbauen lassen.

Einige wichtige Bestandteile sind in *Abbildung 2* dargestellt und folgen dem Sprachschema von Java. So gehören Klassen immer in ein Package. Sowohl die Methode (`JavaCodeUnit`) als auch das Feld (`JavaField`) sind durch ein separates Objekt repräsentiert. Die Benutzungs- und Zugriffsbeziehungen werden durch Objekte vom Typ `JavaMethodCall` beziehungsweise `JavaFieldAccess` abgebildet. Prüfungen auf diesem Objekt-Graph können leicht beschrieben und über eine Definition in Form einer CONDITION an das Lang-API weitergereicht werden. Um diese Struktur besser zu verstehen, sehen wir uns ein einfaches Beispiel an *(siehe Abbildung 3)*: Eine Klasse Agenda befindet sich im Paket `de.test.core` und hat eine Instanzvariable vom Typ `Set` namens `talks`. Auf dieser Variable arbeitet die Methode addTalk, die einen neuen Talk über die `add`-Methode der Klasse `Set` hinzufügt.

Eine – außer als Beispiel – sinnlose Prüfung wäre, sicherzustellen, dass Methoden, die mit `add` beginnen, immer auf ein `Set` zugreifen

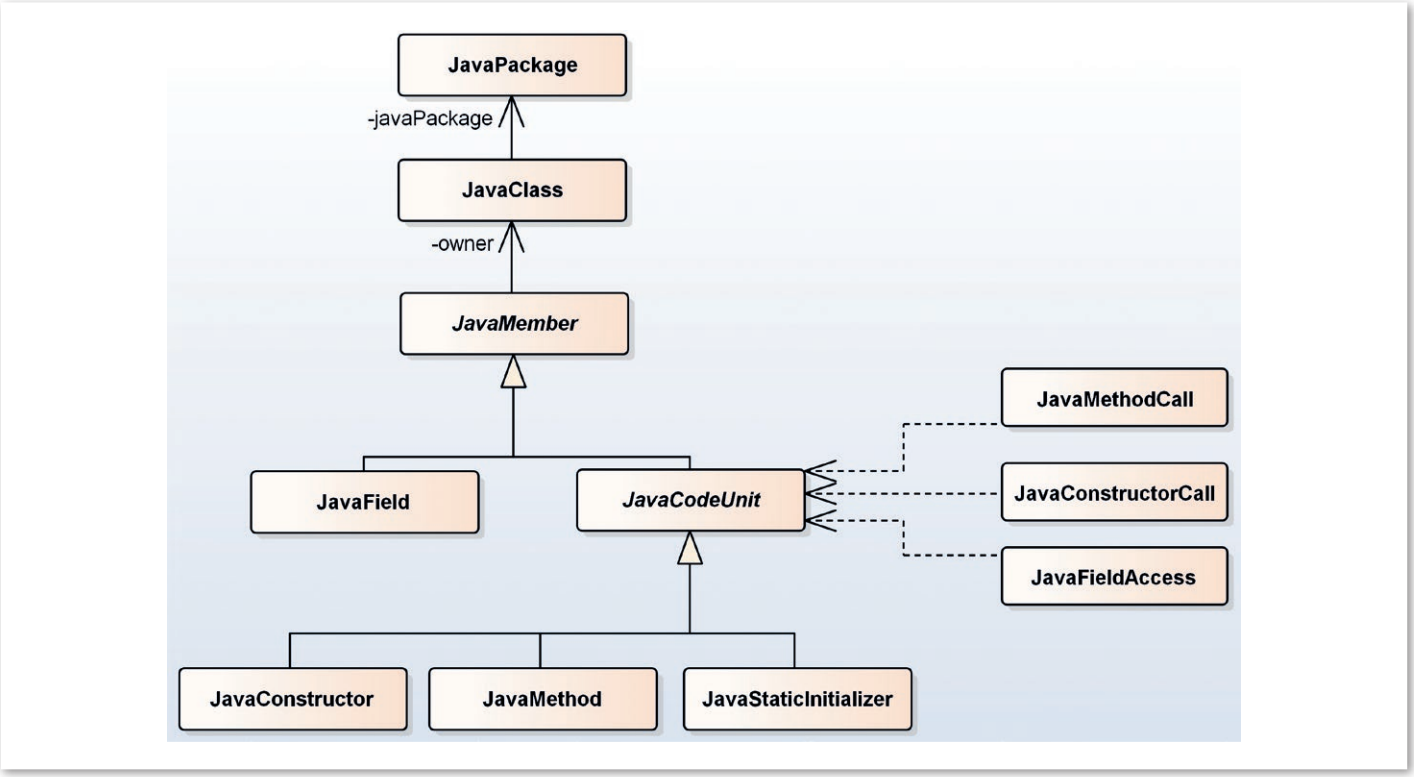



*Abbildung 2: Abbildung des Bytecodes im Core-API (Quelle Thomas Ruhroth)*

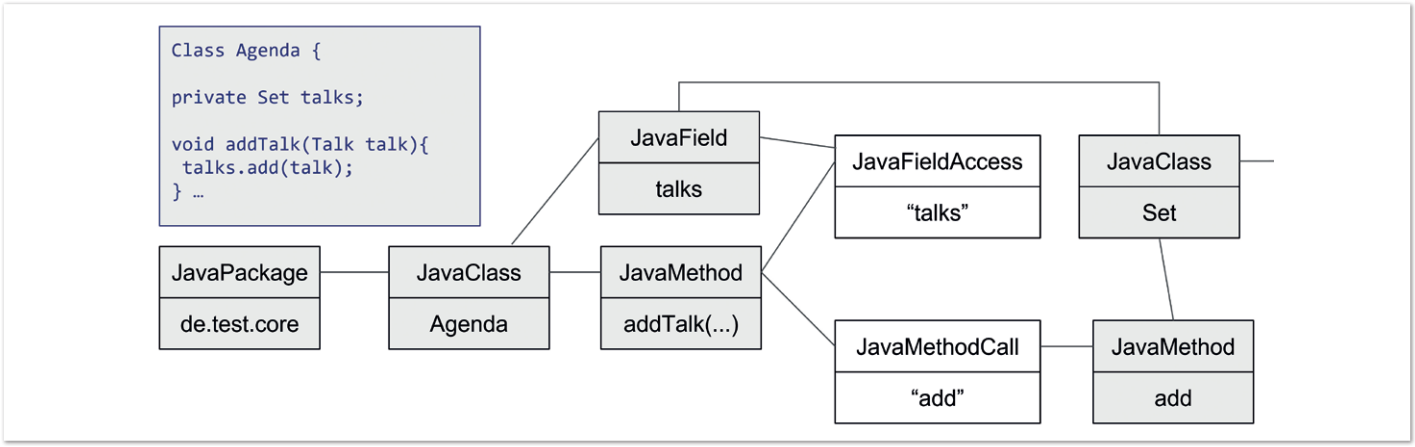



*Abbildung 3: Struktur des Core-Modells für ein Codebeispiel (Quelle Thomas Ruhroth)*

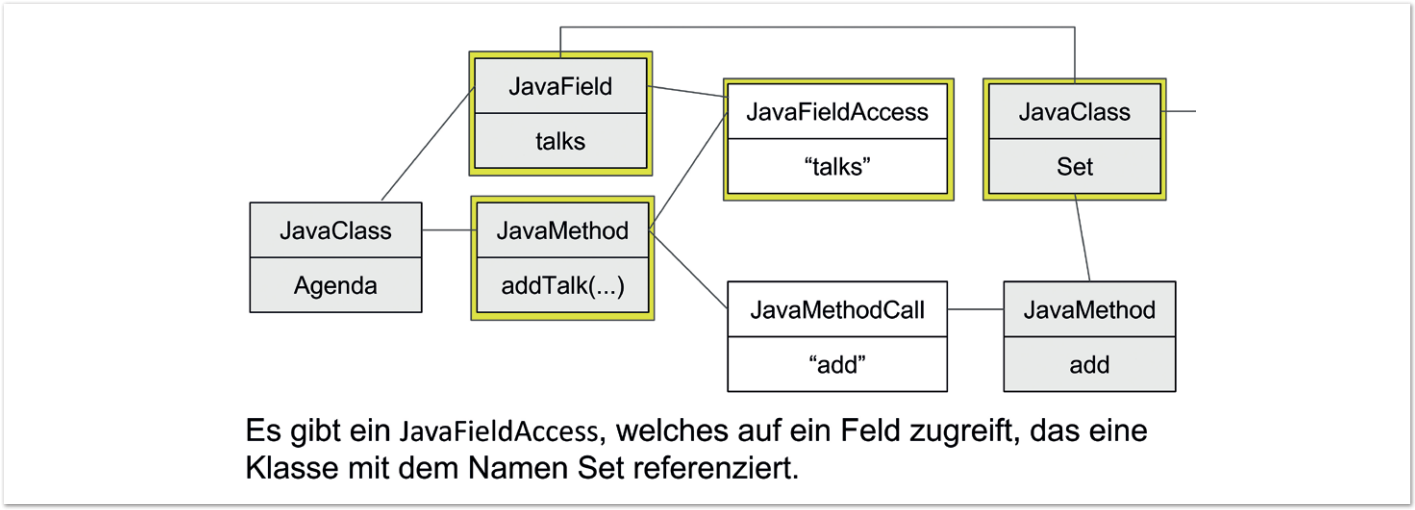



*Abbildung 4: Beispiel einer Core-Regelanalyse (Quelle Thomas Ruhroth)*

sollen. Über den Graphen lässt sich prüfen, dass für alle Methoden-Objekte, deren Name-Property mit add beginnt, folgende Bedingungen gelten sollen: Es gibt ein `JavaFieldAccess`, das auf ein Feld zugreift, das eine Klasse mit dem Namen `Set` referenziert. Im Graphen *(siehe Abbildung 4)* bedeutet das, dass der Zusammenhang der gelb umrandeten Objekte geprüft wird.

## Mit dem Core-API eine Lang-API-Condition erstellen

Mit dem Core-API können Entwickler selbst Prüfungen implementieren und diese anschließend im Lang-API nutzen. Hier werden wir eine vereinfachte Version eines `Deprecated`-Checks aus einem Projekt eines Autors implementieren. Es soll sichergestellt werden, dass Methoden keine als `deprecated` markierten Methoden aufrufen. Ausnahmen bilden Methoden, die selbst als `deprecated` markiert sind. Um dies auf dem Lang-API zu verwirklichen, benötigen wir eine Regel nach dem zuvor vorgestellten Schema. Die Regel müsste eine Form wie aus *Listing 3* aufweisen.

Beim genauen Blick auf das Listing fällt auf, dass im Gegensatz zu *Listing 1* weder nach dem Aufruf von `that` ein Predicate als weitere Methode aufgerufen wird noch beim Aufruf von `should` eine weitere Condition. Das Objekt, das von `that` zurückgegeben wird, kennt keine Methode mit der von uns benötigten Funktion. Ähnlich verhält es sich mit dem Rückgabewert von `should`. Die Rückgabewerte sind Klassen aus dem ArchUnit-Framework. Neue Funktionen für das Lang-API in diesen Klassen könnten über eine Contribution in das ArchUnit-Projekt einfließen. Als Alternative lassen sich selbst implementierte Predicates beziehungsweise Conditions als Parameter von `that` beziehungsweise `should` übergeben.

Der Einstiegspunkt für unsere CONDITION ist das Interface `ArchCondition` *(siehe Listing 4)*. Die zentrale Funktion der `ArchCondition` ist die Methode `check`, die als Parameter die zu prüfende Codestruktur (hier `JavaCodeUnit`) übergeben bekommt. Auf den darauf ausgehenden JavaCalls führen wir die Analyse in der ausgelagerten Methode `checkCall` aus. Das ebenfalls übergebene `ConditionEvents`-Objekt sammelt die Analyseergebnisse.

Für einen `JavaCall` lassen sich sowohl die aufrufende Methode (`getOrigin`) als auch die aufgerufene Methode (`getTarget`) ermitteln. Der Check prüft, ob die aufgerufene Methode oder deren

```
public static final ArchRule DO_NOT_USE_DEPRECATED_OPS
  = methods()
      .that(new NotDeprecatedPredicate())
      .should(new DoNotUseDeprecatedArchCondition());
```

*Listing 3: Verwendung von Predicate und Condition in einer ArchRule*

```
public class DoNotUseDeprecatedArcCondition extends ArchCondition<JavaCodeUnit> {

 public DoNotUseDeprecatedArcCondition (String description, Object... args) {
   super(description, args);
 }

 public DoNotUseDeprecatedArcCondition() {
   this("not call deprecated CodeUnits");
 }

 @Override
 public void check (JavaCodeUnit codeUnit, ConditionEvents conditionEvents) {
   codeUnit.getCallsFromSelf()
       .forEach(call -> checkCall(call, conditionEvents));
 }

 private void checkCall (JavaCall<?> call, ConditionEvents conditionEvents) {
   AccessTarget target = call.getTarget();

   if (target.isAnnotatedWith(Deprecated.class)
       || target.getOwner().isAnnotatedWith(Deprecated.class)) {
          conditionEvents
          .add(SimpleConditionEvent.violated(
             call.getOrigin(),
             "CodeUnit " + call.getOrigin() +
             " calls deprecated " + target.getFullName()));
     }
  }
}
```

*Listing 4: Erstellung einer eigenen Condition*

Klasse `deprecated` ist. Auffälligkeiten werden als Architektur-Verstoß in den `ConditionEvents` abgelegt.

Um diese Prüfung, den Architekturtest, über das Lang-API zur Verfügung zu stellen, wird die Prüfung über eine Subklasse `ArchCondition` implementiert. Über den Typ-Parameter geben wir an, dass unsere `check`-Methode auf der Ebene einer `JavaCodeUnit` eingesetzt wird. Die neue Subklasse muss neben der `check`-Methode einen Konstruktor implementieren. Hierbei genügt es, den Super-Konstruktor aufzurufen. Für den leichteren Aufruf lässt sich zusätzlich ein Konstruktor ohne Parameter deklarieren. Nun kann die über das Core-API entwickelte Prüfung in einer normalen Regel innerhalb des Lang-API verwendet werden *(siehe Listing 3)*.

## Mit dem Core-API ein Lang-API-Predicate erstellen

Das prinzipielle Vorgehen beim Erstellen eines PREDICATE gleicht dem Vorgehen einer CONDITION. Hier wird jedoch eine Subklasse von `DescribedPredicate` gebaut, die einen Wahrheitswert für die Erfüllung der PREDICATES in der Methode `test` berechnet. Über das Predicate möchten wir ausschließlich Methoden selektieren, die nicht selbst oder über die enthaltende Klasse (Owner) als `deprecated` markiert sind *(siehe Listing 5)*.

```
public class NotDeprecatedPredicate extends DescribedPredicate<JavaCodeUnit> {
  public NotDeprecatedPredicate(String description, Object... params) {
     super(description, params);
  }

  public NotDeprecatedPredicate() {
     this("are not deprecated");
  }

  // Hinweis: Vor ArchUnit 1.0.0 heißt die Methode apply
  @Override
  public boolean test(JavaCodeUnit codeUnit) {
     return !codeUnit.isAnnotatedWith(Deprecated.class)
        && !codeUnit.getOwner().isAnnotatedWith(Deprecated.class);
  }
}
```

*Listing 5: Erstellung eines eigenen Predicate*

```
@ArchTest
ArchRule doNotUseDeprecatedOps = MyArchLib.DO_NOT_USE_DEPRECATED_OPS
```

*Listing 6: Verwendung einer selbst definierten ArchRule*

```
@ArchTest
public static final ArchRule configureDeprecatedOpCalls =
  MyArchTestLibrary.deprecationAwareArchitecture()
      .packages("deprecated.callees").areAllowedToBeCalled()
      .packages("deprecated.callers").areAllowedToUseDeprecated();
```

*Listing 7: Verwendung mit parametrisierbaren Regeln*

```
@Override public EvaluationResult evaluate(JavaClasses classes) {
  EvaluationResult result = new EvaluationResult(this, Priority.MEDIUM);
  DeprecatedPredicate predicate = new
      DeprecatedPredicate(packagesAllowedToUseDeprecated);
  DoNotUseDeprecatedArcCondition condition = new
      DoNotUseDeprecatedArcCondition(packagesAllowedToBeCalled);
          result.add(codeUnits().that(predicate).should(condition).evaluate(classes));
  return result;
}
```

*Listing 8: Übergabe der Parameter in das Lang-API*

## Predicate und Condition im Library-API verwenden

Über verschiedenen (Teil-)Projekte oder Module immer die gleichen Regeln zu erstellen ist mühselig. Meist können die Regeln 1:1 übernommen oder eine zusätzliche in leicht abgewandelter Form erstellt werden. Hier hilft es, Prüfungen über das Library-API zusammenzufassen. Die einfachste Variante ist es, wie die `GeneralCodingRules` aus ArchUnit, Regeln als eine Konstante des Typs `ArchRule` bereitzustellen *(siehe Listing 6)*. Die `ArchRule` enthält die Prüfung, wie man sie auch mithilfe des Lang-API schreiben würde. So lässt sich eine Prüfung der `ArchRule` ohne weitere Definition direkt in `ArchTests` nutzen.

Komplexer wird es, wenn man Regeln parametrisieren möchte. Im folgenden Beispiel gehen wir von einer modifizierten Deprecated-Regel aus, die zwei Arten von Ausnahmen bei der Prüfung erlaubt: In der ersten werden Klassen angegeben, auf die trotz Deprecated-Flag zugegriffen werden darf, ohne dass es zu einer Meldung führt – also eine Art Whitelist der aufgerufenen Methoden. Die zweite Ausnahme ist, dass Pakete als Legacy-Pakete deklariert werden, die auf `deprecated`-Methoden zugreifen dürfen – also eine Art Whitelist der aufrufenden Methoden. Die Parametrisierung soll dabei, wie im Library-API von ArchUnit üblich, als Fluent-API gestaltet sein und die Selektierung über die Paketnamen erfolgen *(siehe Listing 7)*.

Ziel dieser Konfiguration ist der Aufbau zweier Listen: Eine Liste der aufrufbaren Pakete und eine Liste der aufrufenden Pakete, um diese später der `NotDeprecatedPredicate` oder der `DoNotUseDeprecatedArchCondition` übergeben zu können. Ähnlich zu den in ArchUnit bereits auf dem Library-API vorhandenen Architekturen definieren wir eine `DeprecationAwareArchitecture`, um dort die beiden Listen zu deklarieren. Der gleichnamige Methodenaufruf gibt uns genau diese Klasse zurück, sodass wir mit den folgenden Methoden darauf arbeiten können. Vorstellbar wäre es, die Methode `areAllowedToBeCalled` direkt innerhalb der Architektur zu deklarieren. Sie könnte die Paketdeklaration über einen Parameter aufnehmen und in die zugehörige Liste speichern *(ähnlich zu Listing 2)*. Allerdings wäre dies nicht besonders sprechend. Daher wird eine Zwischenklasse verwendet, die über den Aufruf von `packages` zurückgegeben wird und die Deklaration für die Übergabe an die Liste aufnimmt. Diese Zwischenklasse ist als innere Klasse der Architektur definiert und kann daher über die darauf definierten Methoden `areAllowedToBeCalled` und `areAllowedToUseDeprecated` direkt in die Listen der Architektur aufnehmen. Die Rückgabe dieser beiden Methoden ist ebenfalls wieder die Architektur, damit die Konfiguration fortgeführt werden kann. Zusätzlich ist die Architektur eine Subklasse von `ArchRule` und kann somit über die Annotation `@ArchTest` als Unittest ausgeführt werden.

Für die Ausführung der ArchRule von ArchUnit muss die Klasse die über das Interface definierten Methoden implementieren. Darunter auch die Methode `evaluate`. Innerhalb dieser Methode lassen sich nun die Konfigurationsparameter an das Predicate und an die Condition innerhalb ihrer Konstruktoren übergeben, sodass diese mit den hierüber neu gewonnenen Informationen die Selektion und die Prüfung ausführen *(siehe Listing 8)*.

In unserem Beispielcode *[4]* lassen sich der dargestellte Ablauf sowie die weitere Auswertung der übergebenen Paketdeklarationen genauer nachvollziehen.

## Fazit

Mit den vorgestellten Techniken lassen sich auf einfache Weise Architekturen für Projekte sicherstellen. In einem Projekt einer der Autoren hat man sich beispielsweise auf eine Auslegung der hexagonalen Architektur geeinigt, die nicht vollständig mit der Definition der von ArchUnit bereitgestellten Onion-Architektur übereinstimmt. Es gibt ein zusätzliches, neben der Architektur stehendes Package zur App-Initialisierung, das in den Adaptern eine zentrale Initialisierung anstößt. Statt diese und einige andere Änderungen in jedem Test neu zu definieren, wurde eine eigene Architektur-Library geschrieben, die neben der angepassten hexagonalen Architektur und dem hier vereinfacht dargestellten Deprecated-Check weitere Prüfungen bereitstellt. Es wird hierdurch eine Vereinheitlichung bei einfacher Einbindung in die Einzelprojekte erreicht.

## Quellen


[1] ArchUnit Homepage und Userguide: *https://www.archunit.org*

[2] Peter Gafert 2018: Java-Architekturen dauerhaft sichern mit

ArchUnit. Java Aktuell 02/2018 *https://www.archunit.org/assets/downloads/02-2018-Java%20aktuell-Java-Architekturen_dauerhaft_sichern_mit_ArchUnit.pdf*

[3] Thomas Ruhroth and Kai Schmidt 2019: Functional core für einen seiteneffektfreien Anwendungskern. Java aktuell 05/2019. *https://www.kai-schmidt.hamburg/wp-content/uploads/05_2019-Java_aktuell-Autor-Thomas_Ruhroth_Kai_Schmidt-Functional_Core_fuer_einen_seiteneffektfreien_Anwendungskern.pdf*

[4] Beispielcode zum Artikel: *https://github.com/electronickai/ArchUnit-Layers-Example*

**Thomas Ruhroth**
msg systems ag
*Thomas.Ruhroth@msg.group*

In seiner industriellen Arbeitserfahrung arbeitete er als Entwickler, Software-Architekt und Business-Analyst in verschieden Bereichen wie Geographische Informationssysteme und Logistik. In der Forschung liegt sein Fachgebiet in der Softwarespezifikation und in der Entwicklung langlebiger Informationssysteme. Der Wissenstransfer aus der Kombination von Forschungsarbeiten mit industrieller Anwendung ist in vielen seiner Projekte eine treibende Kraft.

**Kai Schmidt**
*mail@kai-schmidt.hamburg*

Kai Schmidt ist freiberuflicher Software-Entwickler und -Architekt. Zuvor war er bei den IT-Beratungsunternehmen .msg systems ag und Capgemini angestellt und in seiner über 15-jährigen Projekterfahrung größtenteils in Java und C#-Projekten in den Bereichen Logistik, Flugzeugbau, Finanzen sowie Handel tätig. Er liebt konsistenten Code in den Projekten, auf deren Regeln sich die Teams selbst einigen. Heute berät und beteiligt er sich gerne an betrieblichen Anwendungssystemen und ist in der JUG sowie für Kids4IT und teilweise als Speaker auf Konferenzen und Meetups aktiv.

# Qualität ist echte Teamarbeit!

**Janna Philipp, Anna Platschek, esentri AG**

*Was macht Qualität bei der Produktentwicklung wirklich aus? Warum ist Qualität viel mehr als Testen? Wozu überhaupt Qualität? Was hat Qualität mit der Art und Weise zu tun, wie wir zusammenarbeiten und kommunizieren? Team-Kultur und Qualität, wie passt das zusammen? Wie wirkt sich eine starke Lernkultur auf Qualität aus? Dieser Artikel bietet verschiedene Perspektiven auf Qualität, die über das Testing hinausgehen.*

2001 wurde das Agile Manifest verfasst, um bessere Wege zu erschließen, Software zu entwickeln. Seither lernen Teams und Organisationen, agile Prinzipien und Werte im Alltag der Softwareentwicklung anzuwenden. Oft hören wir von Teams, „das agile Manifest behaupte, dass alles, was auf der rechten Seite steht, das ist, was wir nicht tun sollten": beispielsweise keine Pläne machen, keine Verträge haben, keine Dokumentation schreiben oder Prozesse etablieren.

Schauen wir jedoch genauer hin und nehmen uns die agilen Werte und Prinzipien wirklich zu Herzen, merken wir schnell, dass dies nicht der Fall ist. Vielmehr handelt es sich um eine grundlegend veränderte Art und Weise, wie wir bei der Produktentwicklung zusammenarbeiten, und einen Stil des Projektmanagements, bei dem jede/r im Team gleichermaßen für die Qualität und den Erfolg des Projekts verantwortlich ist – egal, ob Berufsanfänger/in oder Profi! Hoch performante agile Teams wissen, dass jede Entscheidung, die sie bei der Produktentwicklung und im Team treffen, irgendwo auch eine Qualitätsentscheidung ist. In diesem Artikel zeigen wir auf, dass Qualität noch viel mehr als Testen ist und was noch getan werden kann. Ganz nach dem Motto: Qualität ist echte Teamarbeit!

## Kommunikation – ein zentraler Aspekt von Qualität

Arbeiten Sie auch bei einem Entwicklungsunternehmen? Dann können Sie sich folgende Situation bestimmt gut vorstellen: Für eine/n größere/n Auftraggeber/in entwickeln Sie mehrere Komponenten unterschiedlicher Komplexität, die jeweils voneinander unabhängig sind. Sagen wir, in der Größenordnung von 20 verschiedenen Entwicklungsprojekten, die im Aufwand von drei bis 30 Personentagen variieren. Natürlich arbeiten Sie in einem agilen Team innerhalb Ihrer Firma, sodass Sie im Daily immer etwas davon mitbekommen, woran die Kolleginnen und Kollegen gerade arbeiten. Wenn an einer Komponente ein Problem auftritt, schicken Sie oder der/die betreffende Mitarbeitende eine Information an die Kundenfirma mit der Bitte um Klärung und widmen sich dann einer anderen Komponente, in der Hoffnung, dass schon irgendwann eine Rückmeldung kommen wird.

Bald steht eine große Integrationsteststrecke bevor. Diese ist schon lange geplant und kommt trotzdem für die meisten Projektbeteiligten überraschend. Vielleicht werden Sie auch gefragt, wie der Stand bei den Komponenten ist, die Sie mit Ihrem Team entwickeln. Und da merken Sie, wie Sie etwas ins Schwimmen geraten. Vom Gefühl her ist alles irgendwie angefangen, vieles fertiggestellt, aber eine offizielle Abnahme bei der Kundenfirma ist bis jetzt noch nicht durchgeführt worden. Da hat sich ja aber auch keiner gemeldet und die richtigen Ansprechpartner/innen kennen Sie vielleicht auch nicht.

Halten Sie jetzt einen Moment inne. Denken Sie, dass in diesem beschriebenen Kontext eine für das Produkt passende Qualität entsteht? Vielleicht befolgen Sie zwar Coding-Guidelines, haben ein standardisiertes Deployment und eine gute Unittest-Abdeckung. Und trotzdem läuft irgendetwas schief?

In dem obigen Szenario wurde schlichtweg vergessen, dass die Entwicklung von Software ein Ziel verfolgt. Warum wird diese Komponente denn überhaupt benötigt? Um Menschen bei ihren fachlichen Prozessen zu unterstützen. Qualität hat demnach sehr viel mit einem gemeinsamen Verständnis zu tun. Es ist wichtig zu verstehen, was genau die Person, deren fachlicher Prozess unterstützt wird, wirklich braucht. Deshalb gehören auch Personen zum agilen Team, die nicht direkt an der Entwicklung beteiligt sind, etwa Vertreter/innen der betroffenen Fachabteilungen.

Der einzige Weg, ein gemeinsames Verständnis zu schaffen, ist, miteinander zu sprechen. Der direkte verbale Austausch ermöglicht eine enge Feedback-Schleife bis zu dem Punkt, an dem die Beteiligten glauben, sich zu verstehen. So kann funktionierende Software entwickelt werden, die auf einer engen Zusammenarbeit basiert.

Sie haben sicherlich erkannt, dass die richtige Balance vieler verschiedener Faktoren notwendig ist, um Qualität bei der Produktentwicklung zu erzeugen. Hierbei brauchen wir natürlich auch Prozesse. Wir brauchen eine gewisse Dokumentation, um Vereinbarungen festzuhalten und offene Punkte zu visualisieren. Nur sollten Prozesse, Dokumentation und Werkzeuge so eingesetzt werden, dass sie die partnerschaftliche Zusammenarbeit und Kommunikation mit allen Projektbeteiligten unterstützen, wie beim Agilen Manifest beschrieben.

Um auf das obige Szenario zurückzukommen: Fachabteilungen, deren Prozesse unterstützt werden sollen, brauchen hier vor allem Transparenz und eine helfende Hand. Es ist nicht damit getan, guten Code zu schreiben. Die betroffenen Personen sollten in den Entwicklungsprozess mit einbezogen werden. Die Kommunikation darüber, wie die Komponente beispielsweise den fachlichen Prozess unterstützen kann, erzeugt Qualität. Darüber zu kommunizieren, auf welche Entscheidungen des/der Auftraggebenden Sie noch warten oder wo noch Unklarheit besteht, erzeugt auch Qualität. Dies zu visualisieren und sukzessive Hindernisse und Unstimmigkeiten auf dem Weg zu diskutieren, erhöht ebenfalls die Qualität. Letztendlich sollte sich jede/r darauf verlassen können, dass dadurch Dinge ans Licht kommen, die vielleicht mit Mehraufwand, Fehlern oder Konflikten zusammenhängen, jedoch den Mehrwert des Produkts steigern. Denn was nicht kommuniziert und stattdessen unter den Teppich gekehrt wird, kommt in aller Regel später wieder hervor. Und dann wird es meistens richtig unangenehm und teuer.

Aus diesem Grund sind eine gute, offene und transparente Kommunikation sowie das Sichtbarmachen von Informationen für uns wesentlich, um die Qualität bei der Softwareentwicklung zu erhöhen.

## Mindset – wie wirkt es sich auf Qualität aus

Wie aus den Abschnitten zuvor hervorgeht, denken wir, dass Qualität nicht nur viel mehr als Testing ist, sondern auch wirklich alle am Produkt Beteiligten betrifft. Wichtiger als die Auswahl von Methoden und Werkzeugen zur Qualitätsmessung erscheint daher das Bewusstmachen darüber, was Qualität in diesem Setting wirklich bedeutet, was der Qualitätsanspruch ist, wie Qualität erzielt wird und dass wirklich alle wissen, wie Qualität sichergestellt wird. Und auf welche fachlichen Kompetenzen, aber auch Soft Skills es dabei ankommt.

In der Praxis übliche Metriken wie beispielsweise die Fehlerdichte in Bezug auf die Anzahl an Codezeilen haben zwar durchaus ihre Berechtigung. Letztendlich könnte ein/e Entwickler/in die Anzahl der Codezeilen aber auch manipulieren, etwa indem er/sie nach jedem Wort oder gar nach jedem Zeichen einen Zeilenumbruch einfügt. Die Metrik würde sich dadurch verbessern. Die Qualität des Produkts dagegen würde eher leiden, weil ein solcher Code vermutlich deutlich schwerer zu warten wäre.

Was wir sagen wollen ist, dass ein Werkzeug erst dann nützlich ist, wenn es auf eine bestimmte erwünschte Art angewendet wird, bei der alle verstehen, warum und wofür wir es nutzen und wie es die Qualität beeinflusst. Das Ziel von Qualitätsmetriken sollte entsprechend nicht sein, ein möglichst gutes Ergebnis in der Auswertung zu erzielen, sondern Problembereiche transparent zu machen, um Verbesserungspotenzial zu eröffnen.

Wenn Sie nun Produktqualität durch eine Qualitätsfachkraft messen lassen, dann gibt es zwei potenzielle Problempunkte:

1. Qualität ist in diesem Szenario etwas, das im Nachhinein dem Produkt hinzugefügt wird. Der Qualitätsnachweis kommt erst, nachdem die eigentliche Entwicklungsarbeit bereits abgeschlossen ist.
2. Die Verantwortung für Produktqualität wird aus dem Entwicklungsteam herausgenommen und an eine/n Experten/Expertin übertragen, der/die in aller Regel gar nicht direkt am Entwicklungsprozess beteiligt ist, sondern ausschließlich auf Qualität kontrolliert – über verschiedene Produkte hinweg.

Wer kümmert sich nun aber darum, ob eingesetzte Methoden, Vorgehensweisen und Prozesse auch sinnvoll sind? Wo es Verbesserungspotenzial gibt und wo das Team noch besser zusammenarbeiten könnte? Wer identifiziert Konflikte in der Zusammenarbeit und erarbeitet Lösungen? Idealerweise sollte das dort geschehen, wo auch die Entwicklung stattfindet, in Kommunikation mit allen Beteiligten, sodass Entscheidungen über qualitätsverbessernde Maßnahmen auch tragfähig werden. Denn dies ist nur möglich, wenn alle sich dem dahinterliegenden Sinn verpflichten.

Sollten Sie nun wirklich die Produktqualität von der Prozessqualität strikt trennen? Ist es nicht eher so, dass beides eng verwoben ist? Wird ein Team, das selbst für die Qualität seiner Arbeit verantwortlich ist, nur seine Prozesse betrachten? Viel wahrscheinlicher ist, dass im Rahmen einer Reflexion wie oben beschrieben auch Ideen zur Verbesserung am Produkt selbst entstehen, indem das Team zum Beispiel den Einsatz von Tools wie beispielsweise SonarQube *[1]* gemeinsam beschließt.

Das soll allerdings im Umkehrschluss nicht bedeuten, dass jedes Team immer wieder alles neu erfinden muss. Es ist viel wichtiger, bei den Mitarbeiter/innen ein Mindset zu fördern, das dazu führt, dass das Team gemeinsam Verantwortung für die Qualität übernimmt. In diesem Zusammenhang sollte das Team natürlich auch die Möglichkeit haben, Optimierungen nach eigenem Ermessen innerhalb der bestehenden Rahmenbedingungen durchzuführen. So wird Qualität gewissermaßen zu einem Built-in-Feature, anstatt zu einer nachgelagerten Sache, die nur Dinge zurechtbiegt, anstatt sie gleich richtig zu bauen.

Es wird an dieser Stelle klar, wie wichtig die Teamkultur für das Thema Qualität ist. Wie dies durch Feedback und Lernen gefördert wird, erfahren Sie im nächsten Abschnitt.

## Feedback und Lernen – wie erhöht es die Qualität

Wie aus dem Abschnitt zuvor hervorgeht, baut Agilität auf Empirie auf. Das wird auch immer wichtiger, da sich Märkte oft schnell und radikal ändern. Genau genommen leben wir in einem sich ständig ändernden Umfeld mit vielen Unsicherheiten. Sie kennen die Situationen vielleicht auch, in der ein Produkt oder Feature entwickelt wird, das hinterher niemand nutzt oder haben möchte? Oder Tools und Metriken eingeführt werden, die nicht wertbringend sind? Das verursacht nicht nur unnötige Kosten, sondern oft ist die Enttäuschung im Team sehr hoch, da die Arbeit als sinnlos oder nutzlos empfunden wird. Schnell hört man dann Sprüche wie „das interessiert eh niemanden" oder „eigentlich ist es doch eh egal, was wir entwickeln".

Empirie bedeutet nichts anderes als Wissen aus Erfahrung sammeln. Und zwar so früh wie möglich, um reaktionsfähig zu sein, Entscheidungen bewusst und rechtzeitig zu treffen und Chancen wahrzunehmen. Und all das nur, um Kundennutzen und Wert zu stiften. Wir wollen also „Waste" innerhalb von agilen Entwicklungszyklen minimieren.

Hört sich gut an? Dann stellt sich nur noch die Frage, wie Sie und Ihr Team diese empirische Prozesskontrolle schaffen können. Denn mit dem Wissen, das Sie und ihr Team dadurch erlangen, können die darauffolgenden Prozesse stetig verbessert werden und sich den Marktbedingungen immer anpassen – wenn nötig. Damit die immer fortwährende empirische Kontrolle und Verbesserung reibungslos funktionieren, muss auf die drei Säulen der empirischen Prozesskontrolle geachtet werden:

1. **Transparenz:** Nur wer sichtbar und offen für alle arbeitet und während des gesamten Prozesses transparent bleibt, kann erfolgreich und ohne großen (Nach-) Aufwand erfolgreich Ziele und Vorhaben erreichen. Wenn alle Beteiligten stets das „Big Picture" im Blick behalten und dieselbe Sprache sprechen, kann das Team optimal agil arbeiten und bestmögliche Qualität liefern.
2. **Überprüfung/Inspektion:** Bei der Überprüfung des Prozesses können Teams feststellen, wie erfolgreich sie sind und was verbessert oder verändert werden kann und muss – nicht nur in Bezug auf die Fragestellung „wie arbeiten wir zusammen", sondern auch „wie sieht es mit der Qualität aus".
3. **Anpassung/Adaption:** Nachdem der Prozess überprüft wurde, nimmt man die notwendigen Anpassungen vor. Der Prozess,

Qualitätsmaßnahmen oder das Ziel werden angepasst, dadurch verbessert und der agile Lernzyklus kann weitergehen.

Die empirische Prozesskontrolle muss also Teil eines jeden agilen Teams bei der Produktentwicklung sein, um die Qualität so zu sichern, wie es das jeweilige Produkt und der jeweilige Kontext erfordern.

Vielleicht fragen Sie sich an dieser Stelle auch, was genau dafür getan werden kann. Zum einen helfen uns hierbei Retrospektiven und eine starke Feedbackkultur, die zielgerichtet und konstruktiv auf Wachstum abzielen.

## Qualität durch Retrospektiven

Retrospektiven sind Teamtreffen, in denen aus der Vergangenheit gelernt wird. Es wird analysiert, was gut lief, was nicht so gut lief, was gelernt wurde, um so Verbesserungsmaßnahmen für die künftigen Iterationen abzuleiten. Unsere Erfahrung ist jedoch, dass Teams häufig immer wieder die gleichen Dinge diskutieren, sich scheinbar im Kreis drehen und trotzdem bei jeder Retrospektive wieder auf der grünen Wiese beginnen.

Angenommen, ein Team hat schon länger Herausforderungen hinsichtlich der Produktqualität. Dann kann es sinnvoll sein, die nächsten Retrospektiven unter dem Motto: „Wir bekommen unsere Qualität in den Griff" laufen zu lassen. So wird ein klarer Fokus gesetzt und es wird vermieden, immer wieder über die gleichen Themen zu sprechen.

Am Anfang einer solchen Retrospektive können die Ergebnisse der letzten Verbesserungsmaßnahmen zur Qualität auf deren Erfolg untersucht werden, also „überprüft", ob die letzten Verbesserungsmaßnahmen erfolgreich waren. Wenn ja, kann man sich den nächsten Herausforderungen zuwenden. Wenn nein, ist es zuerst einmal sinnvoll, nach den Ursachen zu forschen, um dann darauf basierend die nächsten Maßnahmen zu starten. Sie werden überrascht sein, dass es häufig nicht nur am Testen liegt, sondern beispielsweise auch an Themen wie Kommunikationsschnittstellen, fehlenden Zugriffsrechten zu Tools, unterschiedlichen Qualitäts-Verständnissen oder Intransparenzen im Team liegen kann.

## Qualität durch Feedback

Eine offene Feedbackkultur bringt viele Vorteile. Sie zeigt nicht nur Wertschätzung im Team und ist „Arbeiten auf Augenhöhe", sondern es gibt den Mitarbeiterinnen und Mitarbeitern auch eine Chance, zu lernen, zu wachsen und sich persönlich weiterzuentwickeln – vor allem in Bezug auf Qualität und ein gemeinsames Qualitätsverständnis innerhalb des Teams, im weiteren Sinne sogar innerhalb der gesamten Organisation. Und sind wir doch mal ganz ehrlich, sind Wachstum und Weiterentwicklung nicht ein Grundbedürfnis von uns Menschen, da wir alle einen guten Job machen wollen?

Oft erleben wir es, dass sich Teammitglieder oder Mitarbeitende beschweren:

- „Ich erhalte kaum Feedback"
- „Ich weiß gar nicht, was meine Kolleginnen und Kollegen über unsere Zusammenarbeit denken"
- „Am Ende des Tages weiß ich nicht, ob ich einen guten Job mache oder eben nicht"

Oder es gehen Beschwerden ein wie:

- „Der Code von X ist furchtbar!"
- „Das ist einfach schlecht, was Y abliefert"

Denken Sie doch mal daran, wie oft Sie eine Kollegin oder ein Kollege ärgert, da sie/er etwas getan hat, was nicht Ihrem Qualitätsverständnis entspricht. Fragt man dann, ob dies der Person X oder dem Team mal widergespiegelt wurde, kommt oft ein schnelles „Nein". Irgendwie traurig, oder?

Natürlich muss Feedback auch nicht immer negativ sein, sondern kann auch das Positive hervorheben und so aufzeigen, was man gerne noch mehr in Bezug auf Qualität sehen möchte.

Unsere Erfahrung ist, dass es sehr demotivierend sein kann, kein Feedback zu bekommen und zu geben. Dabei hat es eigentlich nur Vorteile, da Feedback extrem motivierend sein kann und starke Teams aufbaut. Es verstärkt nicht nur Beziehungen, sondern das Verständnis untereinander und verbessert dadurch auch die Zusammenarbeit und Qualität bei der Produktentwicklung. Feedback fördert offene und ehrliche Gespräche und vermeidet dadurch das Aufstauen von Frust. Hierbei sollte Feedback niemals einen Rückschritt bedeuten, sondern immer ein Vorankommen oder das Wachstum eines Individuums, des Teams oder des Produkts bekräftigen. Es sollte nicht einfach „aus Trotz" gegeben werden, sondern eine ehrliche Rückmeldung für die persönliche Weiterentwicklung und Orientierung sein.

## Wie können wir uns Feedback geben?

Hier möchten wir Ihnen noch eine Methode an die Hand geben: Die WWW-Methode, mit der wir sowohl kritisches als auch positives Feedback zeitnah geben können.

**Wahrnehmung**

- Was? Situation und gezeigtes Verhalten konkret beschreiben
- Beispiel: „Mir ist aufgefallen, dass Du in den letzten zwei Wochen Informationen zum Thema XYZ nicht oder erst verzögert an mich weitergeleitet hast."

**Wirkung/Konsequenz**

- Was? Welche Auswirkung/Konsequenzen hat das Verhalten? Wie empfinde ich das gezeigte Verhalten?
- Beispiel: „… mir fehlen dann bei der Bearbeitung wichtige Infos. Das ärgert mich in der Situation, weil ich meine Arbeit nicht abschließen und fertig machen kann."

**Wunsch/Erwartung**

- Was? Gewünschtes/erwartetes Verhalten beschreiben
- Beispiel: „Ich möchte gerne, dass du mir in Zukunft die Informationen zum Thema XYZ zeitnah weiterleitest. Können wir uns darauf einigen?"

## Was wir uns wünschen, das Sie aus diesem Artikel mitnehmen

Bislang steht die Organisations- und Teamkultur noch wenig im Fokus, spielt aber eine bedeutende Rolle bei der Qualität in der Produktent-

wicklung. Wichtig ist ein gemeinsames Verständnis und dass Entwickler/innen sowie Kundinnen und Kunden das Thema ganzheitlich verankern, kennen und mitdenken.

„Was nicht kommuniziert wird und stattdessen unter den Teppich gekehrt wird, kommt in aller Regel später wieder hervor." Deshalb machen wir bei unserer Arbeit immer wieder darauf aufmerksam, dass Qualität nicht nur Testen und guten Code beinhaltet, sondern auch viele Soft und Social Skills erfordert sowie beispielsweise ein Qualitäts-Mindset, Feedback geben und nehmen, Kommunikationsfähigkeit, transparentes Arbeiten, Teamfähigkeit und -arbeit, Team-Werte und -Prinzipien leben und eine hohe Reflexionsfähigkeit, verstehen, warum und wofür wir das Ganze machen. Dabei sollten auch die Bedürfnisse der Kundinnen und Kunden nicht vergessen werden.

Qualität sollte kein nachgelagerter Anhang sein, sondern „built-in". Jede/r ist verantwortlich für gute Qualität und sollte über den eigenen Tellerrand hinaus mitdenken.

## Referenzen

[1] *https://www.sonarqube.org/*

**Janna Philipp**
esentri AG
*info@esentri.com*

Janna Philipp ist fasziniert von den Arbeitsweisen, die den Menschen in den Fokus setzen, Selbstständigkeit und Verantwortungsbewusstsein sowie persönliche und teambasierte Entfaltung fördern. Mit viel Leidenschaft, neuen Impulsen und Inspiration sowie einer systemischen Sicht auf die Dinge entwickelt Janna Philipp gemeinsam mit den anderen Kolleginnen und Kollegen innerhalb des Geschäftskreises „Digitale Kultur" bei der esentri AG, Organisationen, Teams und Menschen bei verschiedenen Kundinnen und Kunden. Für eine nachhaltige Zukunft!

**Anna Platschek**
esentri AG
*info@esentri.com*

Aus ihrer mehrjährigen Erfahrung in vielfältigen IT-Projekten konnte Frau Platschek eine signifikante Gemeinsamkeit ableiten: Unabhängig vom spezifischen Kontext war eine gelungene Kommunikation der Schlüssel zum Projekterfolg und der Güte der gelieferten Qualität. Sie legt großen Wert darauf, Grauzonen und potenzielle Verständigungsprobleme durch zielgerichtete Kommunikation mit den richtigen Personen zur richtigen Zeit aufzulösen, sodass ein gemeinsames Verständnis über Projekt und Ziele aufgebaut wird.


## Mitglieder des iJUG

01 Android User Group Düsseldorf
02 BED-Con e.V.
03 Clojure User Group Düsseldorf
04 DOAG e.V.
05 EuregJUG Maas-Rhine
06 JUG Augsburg
07 JUG Berlin-Brandenburg
08 JUG Bremen
09 JUG Bielefeld
10 JUG Bonn
11 JUG Darmstadt
12 JUG Deutschland e.V.
13 JUG Dortmund
14 JUG Düsseldorf rheinjug
15 JUG Erlangen-Nürnberg
16 JUG Freiburg
17 JUG Goldstadt
18 JUG Görlitz
19 JUG Hannover
20 JUG Hessen
21 JUG HH
22 JUG Ingolstadt e.V.
23 JUG Kaiserslautern
24 JUG Karlsruhe
25 JUG Köln
26 Kotlin User Group Düsseldorf
27 JUG Mainz
28 JUG Mannheim
29 JUG München
30 JUG Münster
31 JUG Oberland
32 JUG Ostfalen
33 JUG Paderborn
34 JUG Passau e.V.
35 JUG Saxony
36 JUG Stuttgart e.V.
37 JUG Switzerland
38 JSUG
39 Lightweight JUG München
40 SOUG e.V.
41 SUG Deutschland e.V.
42 JUG Thüringen
43 JUG Saarland

iJUG Verbund

www.ijug.eu

## Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Melanie Andrisek, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Bildnachweis:
Titel: Bild © freepik
*https://freepik.com*
S. 10 + 11: Bild © fullvector
*https://freepik.com*
S. 12 + 13: Bild © freepik
*https://freepik.com*
S. 20: Bild © storyset
*https://freepik.com*
S. 24 + 25: Bild © freepik
*https://freepik.com*
S. 32 + 33: Bild © vectorjuice
*https://freepik.com*
S. 40 + 41: Bild © rawpixel.com
*https://freepik.com*

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org

Mediadaten und Preise:
*www.doag.org/go/mediadaten*

Druck:
WIRmachenDRUCK GmbH
*www.wir-machen-druck.de*



## Inserentenverzeichnis

# Die Oracle-
# Anwenderkonferenz


Eventpartner: AOUG AUSTRIAN ORACLE USER GROUP SOUG swiss oracle user group

anwenderkonferenz.doag.org